Week 6: Bike Sharing in Washington D.C.

Using RNN to Build a Time Series Forecasting Model

Part 1: Background Information

Abstract

Bike sharing systems have emerged as a modern solution to traditional bike rentals, automating the rental process and enabling users to conveniently access bicycles from one location and return them to another. With a global presence encompassing over 500 programs and half a million bicycles, these systems play a pivotal role in addressing traffic congestion, environmental sustainability, and public health concerns. Moreover, the data generated by bike sharing systems presents a rich source for research, with detailed records of travel durations and locations offering insights into urban mobility patterns.

In this project, we explore the application of Recurrent Neural Networks (RNNs) for time series forecasting in bike sharing systems. RNNs are well-known for their ability to effectively model sequential data, making them ideal for tasks such as time series prediction. By leveraging RNNs, we aim to develop predictive models that can forecast bike rental demand with high accuracy. Through experimentation and analysis, we seek to demonstrate the effectiveness of RNNs in capturing the complex temporal dynamics of bike sharing data, thereby contributing to the advancement of predictive analytics in urban transportation systems.

Introduction:

Bike sharing systems represent a contemporary evolution of traditional bike rentals, revolutionizing the process by automating membership, rental, and return procedures. These systems enable users to effortlessly rent a bike from one location and return it to another. With over 500 bike-sharing programs worldwide, boasting a collective fleet of more than 500,000 bicycles, these systems have garnered significant attention for their pivotal role in addressing traffic congestion, environmental sustainability, and public health concerns.

Beyond their practical applications, the data generated by bike sharing systems has sparked keen interest among researchers. Unlike other modes of transportation such as buses or subways, bike sharing systems meticulously document travel durations, as well as departure and arrival locations. This unique characteristic transforms bike sharing networks into virtual sensor arrays capable of capturing mobility patterns within urban landscapes. Consequently, analysts anticipate that by monitoring these datasets, it will be possible to detect and analyze crucial urban events, offering insights into city dynamics that were previously inaccessible.

Data Description:

The rental dynamics within bike-sharing systems are intricately tied to environmental and seasonal variables. Factors such as weather conditions, precipitation, day of the week, season, and time of day exert significant influence on rental behaviors.

For this study, we utilized the core dataset comprising a two-year historical log spanning the years 2011 and 2012 from the Capital Bikeshare system in Washington D.C., USA. This dataset is publicly available via the Capital Bikeshare system website at [http://capitalbikeshare.com/system-data (http://capitalbikeshare.com/system-data)](http://capitalbikeshare.com/system-data).

In order to comprehensively analyze the rental patterns, we aggregated the data both on a two-hourly and daily basis. Additionally, we enriched the dataset by integrating relevant weather and seasonal information. Weather data was sourced from [http://www.freemeteo.com (http://www.freemeteo.com)](http://www.freemeteo.com) to augment our analysis and provide contextual insights into the rental trends observed in the Capital Bikeshare system.

Dataset:

The dataset is composed of two .csv files: hours.csv and day.csv. Both hour.csv and day.csv have the following fields, except hr which is not available in day.csv:

- instant: record index
- dteday : date
- season : season (1:springer, 2:summer, 3:fall, 4:winter)
- yr : year (0: 2011, 1:2012)
- mnth : month (1 to 12)
- hr : hour (0 to 23)

- holiday : weather day is holiday or not (extracted from http://dchr.dc.gov/page/holiday-schedule (http://dchr.dc.gov/page/holiday-schedule))
- weekday : day of the week
- workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
- weathersit :
  - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
  - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp : Normalized temperature in Celsius. The values are divided to 41 (max)
- atemp: Normalized feeling temperature in Celsius. The values are divided to 50 (max)
- hum: Normalized humidity. The values are divided to 100 (max)
- windspeed: Normalized wind speed. The values are divided to 67 (max)
- casual: count of casual users
- registered: count of registered users
- cnt: count of total rental bikes including both casual and registered

The dataset is available on Kaggle at: https://www.kaggle.com/code/hozler/time-series-forecasting-with-rnn/input

Project Objective:

The primary objective of this project is to develop predictive models for bike rental count, both hourly and daily, leveraging environmental and seasonal variables. Specifically, we aim to achieve the following tasks:

Regression: Utilize machine learning techniques to predict the bike rental count on an hourly or daily basis, considering factors such as weather conditions, seasonality, and temporal patterns.

Event and Anomaly Detection: Investigate the correlation between bike rental counts and significant events within the town, which can be identified through search engine queries. By validating against known events such as Hurricane Sandy, we aim to develop algorithms for event detection and anomaly identification using the bike sharing dataset.

Through these tasks, we seek to demonstrate the predictive capabilities of machine learning models in capturing the complex dynamics of bike rental behaviors, as well as contribute to the development of event detection algorithms for urban systems.

Evaluation Plan:

1. Evaluation Metrics:

- **Mean Absolute Error (MAE):** The primary metric to evaluate the predictive accuracy of the models. Lower MAE indicates better performance.
- **MAE/Mean Ratio:** Provides insight into the relative error compared to the mean value of the target variable. A lower ratio signifies better performance.
- **Correctness:** Complementary metric derived from MAE/Mean Ratio, representing the percentage of correctness. Higher correctness indicates better performance.

2. Data Split:

- **Training Data:** Historical bike sharing data spanning a defined period (e.g., 2011-2012) used for model training.
- **Validation Data:** A portion of the dataset reserved for tuning hyperparameters and evaluating model performance during training.
- **Hold-Out Data:** A separate subset of data not used during model training or validation, reserved for final model evaluation to assess real-world performance.

3. Evaluation Procedure:

1. Data Preprocessing:

- Standardize features: Normalize numerical features to a common scale.
- Handle missing values: Impute or remove missing values appropriately.
- Feature engineering: Create new features if necessary, such as time-based features or weather interactions.

2. Model Training:

- Train multiple models, including Model 1, Model 2, and Model 3, using appropriate algorithms (e.g., RNNs).
- Tune hyperparameters using the validation data to optimize model performance.

3. Model Evaluation:

- Evaluate each model's performance using the validation data based on the defined evaluation metrics.
- Select the best-performing model based on MAE, MAE/Mean Ratio, and Correctness metrics.

4. Final Model Selection:

- Validate the selected model using the hold-out data to ensure generalization and real-world performance.
- Compare the model's performance on the hold-out data with validation results to confirm consistency.

5. Performance Analysis:

- Analyze the predictive accuracy and correctness of the selected model compared to other models.
- Investigate the reasons behind the superior performance of the selected model, considering factors such as architecture, hyperparameters, and feature engineering.

4. Reporting and Documentation:

- **Final Report:** Summarize the evaluation process, including data preprocessing, model training, and evaluation. Present detailed results of each model's performance based on MAE, MAE/Mean Ratio, and Correctness metrics. Provide insights into the reasons for the selected model's superiority and its implications for the project objectives.
- **Documentation:** Document all data preprocessing steps, hyperparameters, and model configurations for reproducibility. Include visualizations, tables, and figures to illustrate key findings and comparisons.

5. Continuous Improvement:

- **Model Refinement:** Periodically retrain the selected model with updated data to maintain relevance and accuracy. Explore advanced techniques or alternative algorithms to further improve model performance.
- **Feedback Mechanism:** Establish a feedback mechanism to gather insights from stakeholders and end-users for model refinement and future iterations.

Acknowledgments:

This dataset was created by Haid Fanaee-T and Joao Gama for their paper *Event labeling combining ensemble detectors and background knowledge.*

Fanaee-T, Hadi, and Gama, Joao, "Event labeling combining ensemble detectors and background knowledge", Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg, doi:10.1007/s13748-013-0040-3.

Available at: https://www.kaggle.com/code/hozler/time-series-forecasting-with-rnn/input (https://www.kaggle.com/code/hozler/time-series-forecasting-with-rnn/input)

Importing the necessary libraries

In [153]:

```
# Import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os

#From numpy
from numpy import array, hstack

#From pandas
from pandas.plotting import lag_plot

#From sklearn
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV

#From keras
from keras.models import Sequential
from keras.layers import Dense, GRU, LSTM, RNN, SimpleRNN
from keras.preprocessing.sequence import TimeseriesGenerator
from keras.layers import Dropout
from keras.optimizers import Adam
from keras.layers.core import Activation
from keras.callbacks import LambdaCallback
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor

#From tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.optimizers import Adam
```

Reading the dataset

In [154]:

```
# Get the current working directory
current_directory = os.getcwd()

# Define path to the day.csv file
dataset_path = os.path.join(current_directory, 'day.csv')

# Read the dataset
dataset = pd.read_csv(dataset_path)
```

Project Summary

Project Summary:

The project focuses on time series forecasting using Recurrent Neural Networks (RNNs), specifically exploring the effectiveness of different RNN architectures including Simple RNNs, Gated Recurrent Units (GRUs), and Long Short-Term Memory (LSTM) networks. By capitalizing on the inherent sequential data processing capabilities of RNNs, the objective is to develop robust models capable of accurately predicting future trends in time series data. This comprehensive approach involves various stages such as data preprocessing, feature engineering, model architecture design, hyperparameter tuning, model training, and performance evaluation against established metrics.

The dataset used in the project comprises historical records spanning multiple temporal dimensions, allowing for the exploration of complex temporal patterns and dependencies. Through rigorous experimentation and analysis, the project aims to uncover insights into the strengths and weaknesses of different RNN architectures in handling diverse time series forecasting tasks.

Additionally, the project extends beyond basic model development to encompass advanced techniques such as sensitivity analysis and model comparison. By systematically varying input parameters and evaluating model performance across different scenarios, the project seeks to provide a comprehensive understanding of the robustness and sensitivity of each RNN architecture. Furthermore, comparing the performance of Simple RNNs, GRUs, and LSTMs allows for insightful observations regarding the impact of architectural complexities on predictive accuracy and computational efficiency.

Overall, the project aspires to contribute to the advancement of predictive analytics across various domains including finance, weather

Part II: Exploratatory Data Analysis

About the Data

In [155]:
```
dataset.head()
```

Out[155]:

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	regis
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331	
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131	
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120	
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	0.212122	0.590435	0.160296	108	
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	0.229270	0.436957	0.186900	82	

In [156]:
```
dataset.shape
```

Out[156]: (731, 16)

Summary Statistics

Thw following code will generate summary statistics including count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile or Q2), 75th percentile (Q3), and maximum for each numerical variable in your dataset. It will provide insights into the central tendency and variability of your data.

In [157]:
```
# Calculate summary statistics for numerical variables
summary_stats = dataset.describe()

# Print the summary statistics
print(summary_stats)
```

```
          instant      season          yr        mnth     holiday     weekday  \
count  731.000000  731.000000  731.000000  731.000000  731.000000  731.000000
mean   366.000000    2.496580    0.500684    6.519836    0.028728    2.997264
std    211.165812    1.110807    0.500342    3.451913    0.167155    2.004787
min      1.000000    1.000000    0.000000    1.000000    0.000000    0.000000
25%    183.500000    2.000000    0.000000    4.000000    0.000000    1.000000
50%    366.000000    3.000000    1.000000    7.000000    0.000000    3.000000
75%    548.500000    3.000000    1.000000   10.000000    0.000000    5.000000
max    731.000000    4.000000    1.000000   12.000000    1.000000    6.000000

       workingday  weathersit        temp       atemp         hum   windspeed  \
count  731.000000  731.000000  731.000000  731.000000  731.000000  731.000000
mean     0.683995    1.395349    0.495385    0.474354    0.627894    0.190486
std      0.465233    0.544894    0.183051    0.162961    0.142429    0.077498
min      0.000000    1.000000    0.059130    0.079070    0.000000    0.022392
25%      0.000000    1.000000    0.337083    0.337842    0.520000    0.134950
50%      1.000000    1.000000    0.498333    0.486733    0.626667    0.180975
75%      1.000000    2.000000    0.655417    0.608602    0.730209    0.233214
max      1.000000    3.000000    0.861667    0.840896    0.972500    0.507463

            casual   registered          cnt
count   731.000000   731.000000   731.000000
mean    848.176471  3656.172367  4504.348837
std     686.622488  1560.256377  1937.211452
min       2.000000    20.000000    22.000000
25%     315.500000  2497.000000  3152.000000
50%     713.000000  3662.000000  4548.000000
75%    1096.000000  4776.500000  5956.000000
max    3410.000000  6946.000000  8714.000000
```

- Count: There are 731 observations in the dataset for each variable, indicating that there are no missing values.
- Mean: The mean value represents the average value of each variable across all observations.
- Standard Deviation (Std): The standard deviation measures the dispersion or variability of the values from the mean. It indicates how spread out the values are.
- Minimum (Min): The minimum value is the smallest observed value for each variable.
- 25th Percentile (Q1): The 25th percentile (Q1) represents the value below which 25% of the observations fall.
- Median (50th Percentile or Q2): The median is the middle value of the dataset when it is ordered from least to greatest. It separates the higher half from the lower half of the data.
- 75th Percentile (Q3): The 75th percentile (Q3) represents the value below which 75% of the observations fall.
- Maximum (Max): The maximum value is the largest observed value for each variable.

Checking for Missing Values

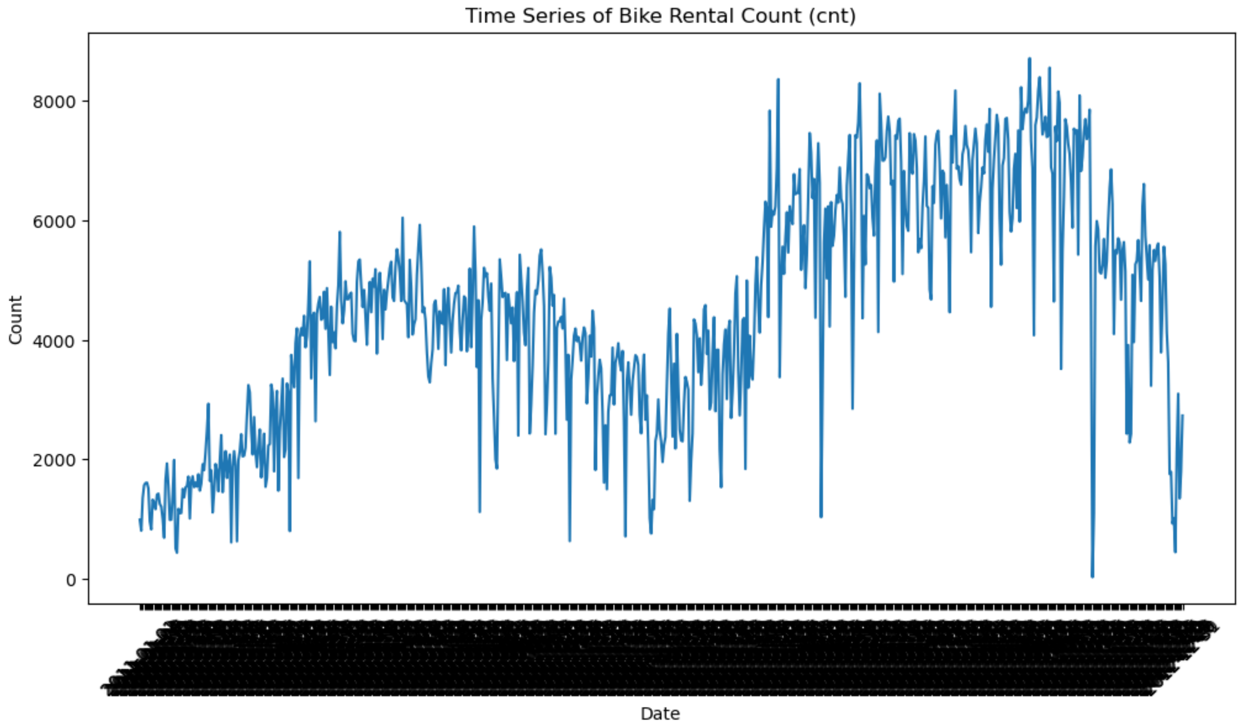
In [158]:

```python
# Check for missing values
missing_values = dataset.isnull().sum()

# Print the number of missing values for each column
print("Missing Values:")
print(missing_values)
```

```
Missing Values:
instant       0
dteday        0
season        0
yr            0
mnth          0
holiday       0
weekday       0
workingday    0
weathersit    0
temp          0
atemp         0
hum           0
windspeed     0
casual        0
registered    0
cnt           0
dtype: int64
```

Time Series Plot

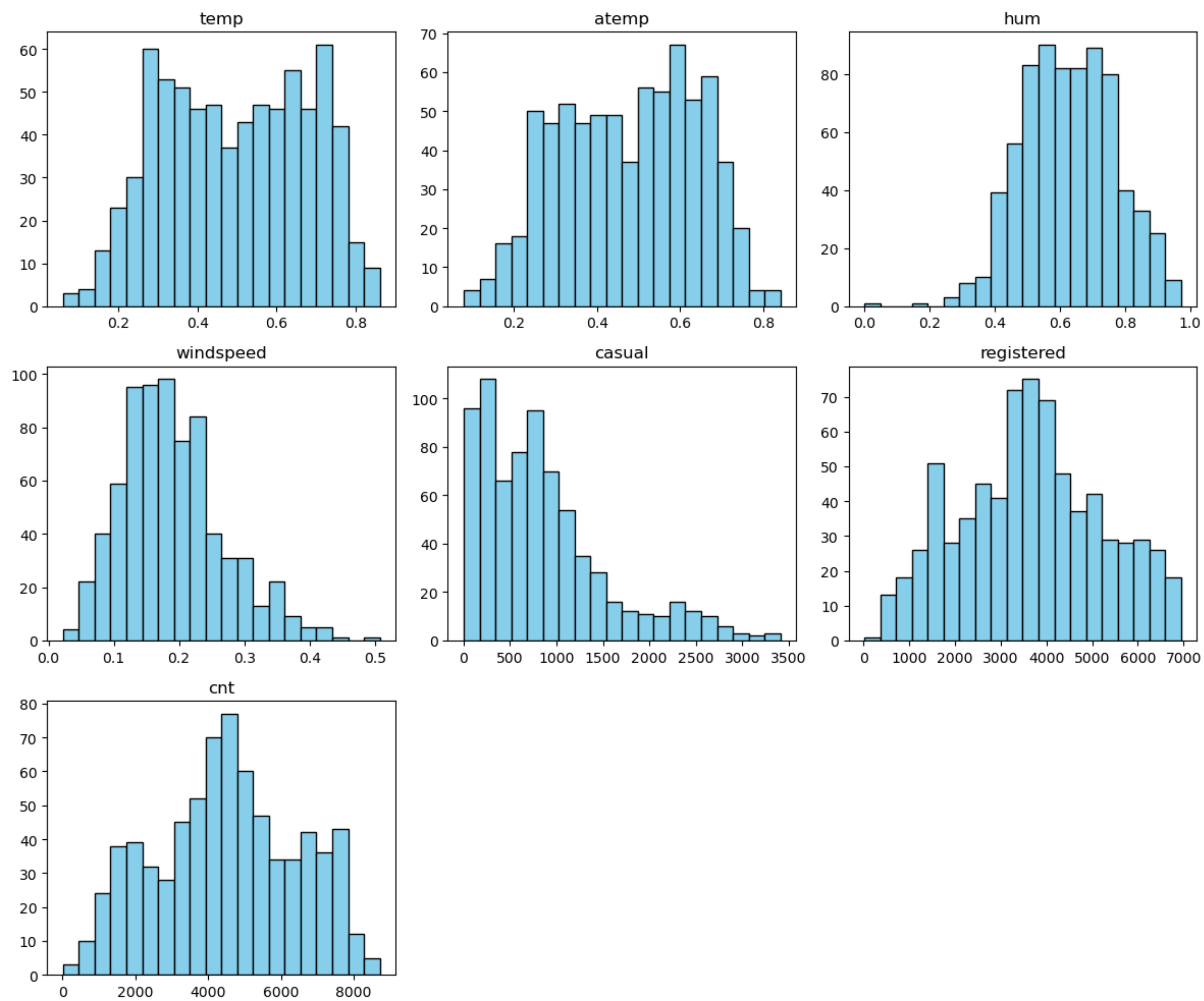
In [159]:
```
# Time Series Plot
plt.figure(figsize=(12, 6))
plt.plot(dataset['dteday'], dataset['cnt'])
plt.title('Time Series of Bike Rental Count (cnt)')
plt.xlabel('Date')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



Distribution Plots

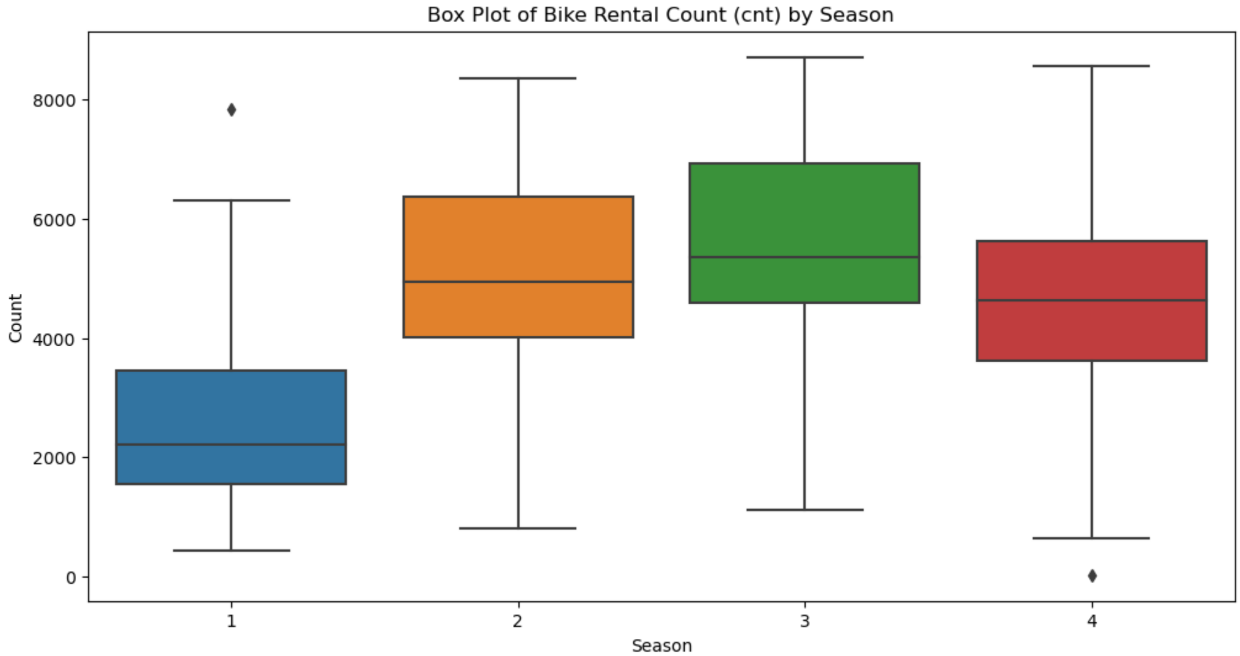
In [160]:

```
# Histograms
numerical_vars = ['temp', 'atemp', 'hum', 'windspeed', 'casual', 'registered', 'cnt']
plt.figure(figsize=(12, 10))
for i, var in enumerate(numerical_vars):
    plt.subplot(3, 3, i + 1)
    plt.hist(dataset[var], bins=20, color='skyblue', edgecolor='black')
    plt.title(var)
plt.tight_layout()
plt.show()
```



Box Plots

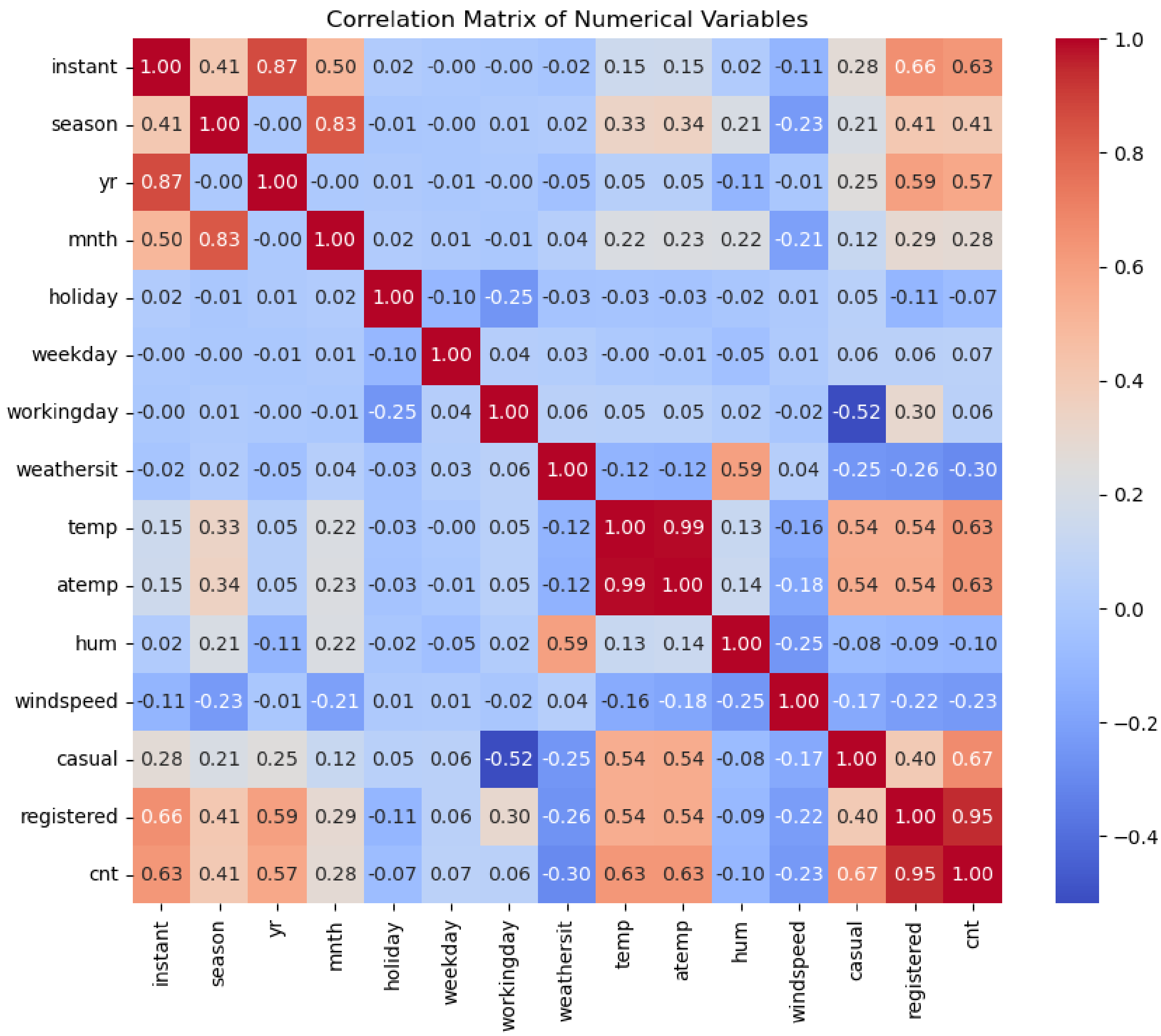
In [161]:
```
# Box Plots
plt.figure(figsize=(12, 6))
sns.boxplot(x='season', y='cnt', data=dataset)
plt.title('Box Plot of Bike Rental Count (cnt) by Season')
plt.xlabel('Season')
plt.ylabel('Count')
plt.show()
```



Correlation Matrix

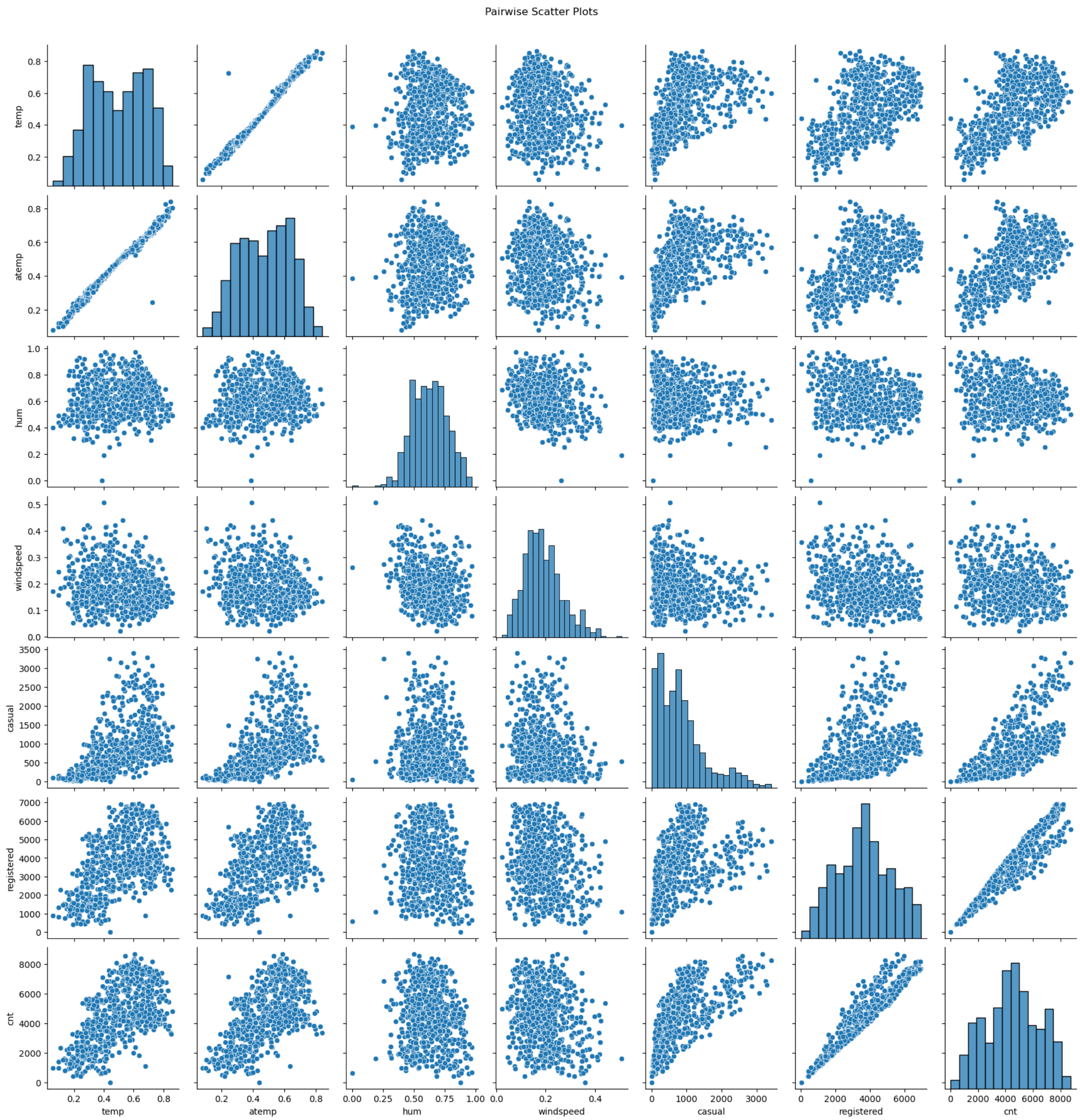
In [162]:

```
# Correlation Matrix
plt.figure(figsize=(10, 8))
correlation_matrix = dataset.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of Numerical Variables')
plt.show()
```



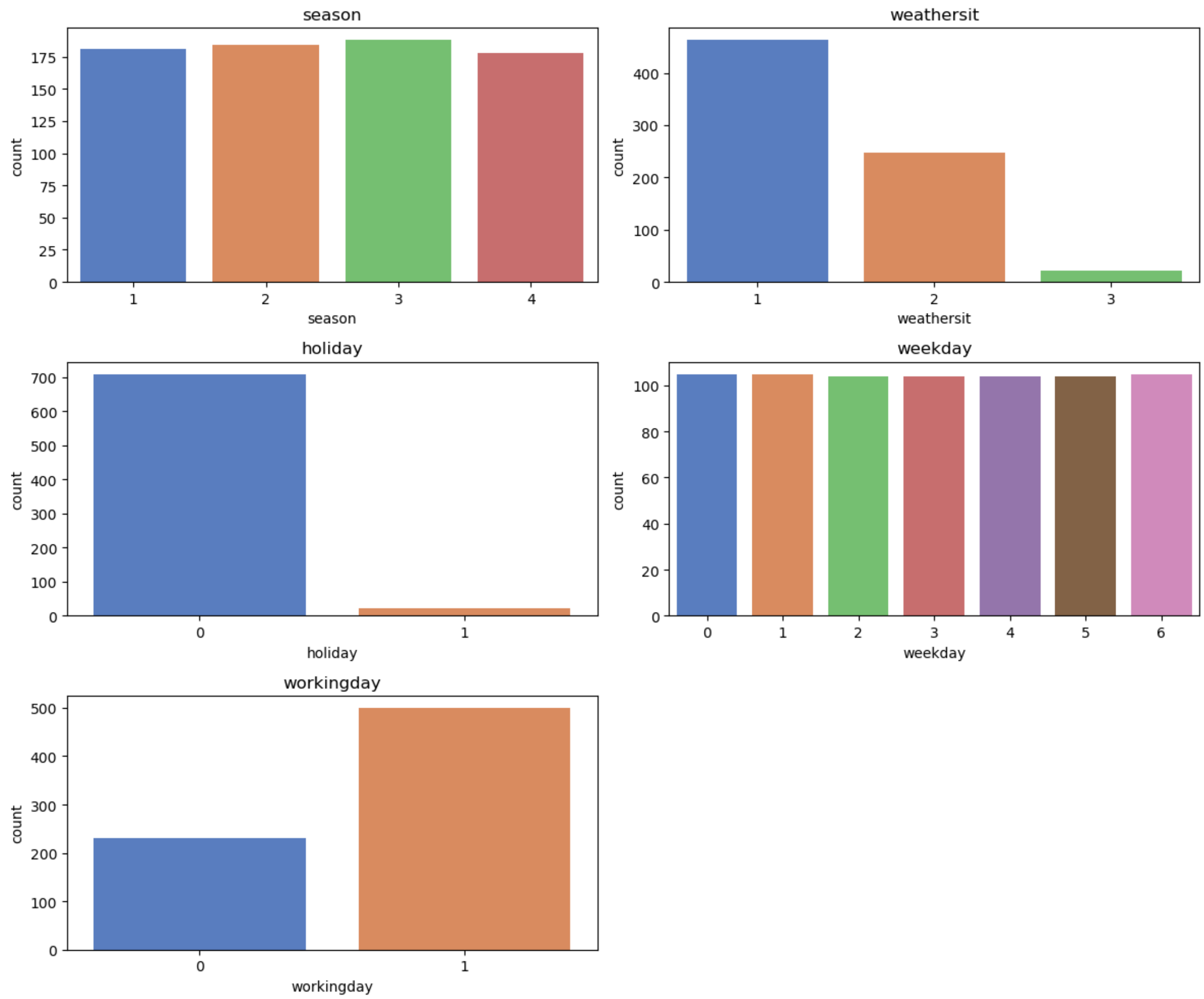
Scatter Plots

In [163]:
```
# Scatter Plots
sns.pairplot(dataset[['temp', 'atemp', 'hum', 'windspeed', 'casual', 'registered', 'cnt']])
plt.suptitle('Pairwise Scatter Plots', y=1.02)
plt.show()
```



Bar Plots

In [164]:
```
# Bar Plots
categorical_vars = ['season', 'weathersit', 'holiday', 'weekday', 'workingday']
plt.figure(figsize=(12, 10))
for i, var in enumerate(categorical_vars):
    plt.subplot(3, 2, i + 1)
    sns.countplot(x=var, data=dataset, palette='muted')
    plt.title(var)
plt.tight_layout()
plt.show()
```

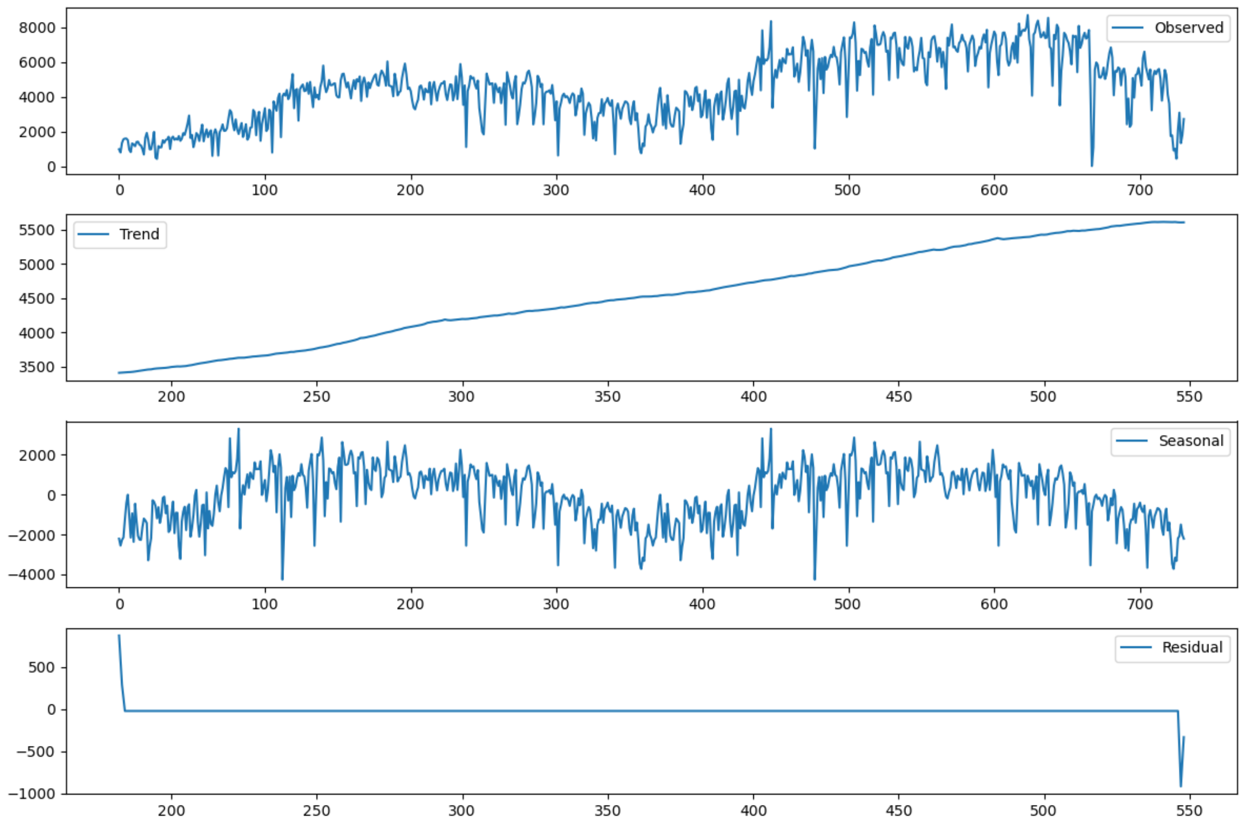


Seasonal Decomposition Plots

- First, convert the 'dteday' column to datetime type and set it as the index of the DataFrame to facilitate time series analysis.
- Then, use the seasonal_decompose function from the statsmodels library to decompose the time series into its trend, seasonal, and residual components. We specify the model as 'additive' and set the period parameter to 365 for yearly seasonality.
- Finally, we plot the observed time series, trend, seasonal, and residual components using matplotlib.

In [165]:

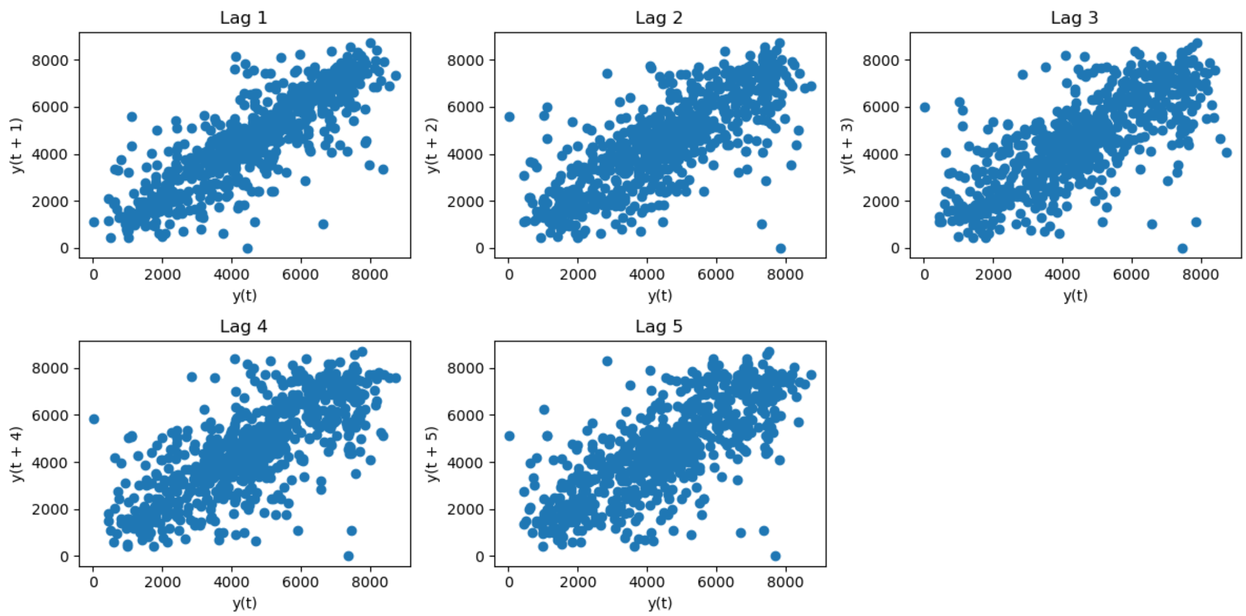
```
# Convert 'dteday' column to datetime type
dataset['dteday'] = pd.to_datetime(dataset['dteday'])

# Perform seasonal decomposition
decomposition = sm.tsa.seasonal_decompose(dataset['cnt'], model='additive', period=365)

# Plot the decomposition
plt.figure(figsize=(12, 8))
plt.subplot(4, 1, 1)
plt.plot(decomposition.observed, label='Observed')
plt.legend()
plt.subplot(4, 1, 2)
plt.plot(decomposition.trend, label='Trend')
plt.legend()
plt.subplot(4, 1, 3)
plt.plot(decomposition.seasonal, label='Seasonal')
plt.legend()
plt.subplot(4, 1, 4)
plt.plot(decomposition.resid, label='Residual')
plt.legend()
plt.tight_layout()
plt.show()
```



Lag Plots

- First, the target variable is specified to create lag plots (e.g., 'cnt').
- Lag plots are created for the target variable using a loop over different lag values.

In [166]:
```
# Select the target variable for which you want to create lag plots (e.g., 'cnt')
target_variable = 'cnt'

# Create lag plots for the target variable
plt.figure(figsize=(12, 6))
for lag in range(1, 6):  # You can adjust the range of lags as needed
    ax = plt.subplot(2, 3, lag)
    lag_plot(dataset[target_variable], lag=lag)
    ax.set_title(f'Lag {lag}')
plt.tight_layout()
plt.show()
```



Part III: Data Preprocessing

Transforming Certain Features into Categorical Data

One Hot Encoding willbe applied for categorical features. In this case weekday, weathersit and mnth features categorical in nature and should have One Hot Encoding applied.

In [167]:
```
one_hot = pd.get_dummies(dataset['weekday'], prefix='weekday')
dataset = dataset.join(one_hot)
```

In [168]:
```
one_hot = pd.get_dummies(dataset['weathersit'], prefix='weathersit')
dataset = dataset.join(one_hot)
```

In [169]:
```
one_hot = pd.get_dummies(dataset['mnth'], prefix='mnth')
dataset = dataset.join(one_hot)
```

In [170]:
```
dataset.head()
```

Out[170]:

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	...	mnth_3	mnth_4	mnth_5	mnth_6	mnth_
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	...	0	0	0	0	
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	...	0	0	0	0	
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	...	0	0	0	0	
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	...	0	0	0	0	
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	...	0	0	0	0	

5 rows × 38 columns

Applying Scaling to the Data

It appears as though all of the features have previously been scaled by the authors, save for the feature cnt, which happens to by our Y value. Therefore, we will only apply scaling to this feature.

In [171]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(array(dataset['cnt']).reshape(len(dataset['cnt']), 1))
series = pd.DataFrame(scaled)
series.columns = ['cntscl']
```

In [172]:
```
series.head()
```

Out[172]:

	cntscl
0	0.110792
1	0.089623
2	0.152669
3	0.177174
4	0.181546

In [173]:
```
dataset = pd.merge(dataset, series, left_index=True, right_index=True)
```

In [174]:
```
dataset.head()
```

Out[174]:

mnth	holiday	weekday	workingday	weathersit	temp	...	mnth_4	mnth_5	mnth_6	mnth_7	mnth_8	mnth_9	mnth_10	mnth_11	r
1	0	6	0	2	0.344167	...	0	0	0	0	0	0	0	0	
1	0	0	0	2	0.363478	...	0	0	0	0	0	0	0	0	
1	0	1	1	1	0.196364	...	0	0	0	0	0	0	0	0	
1	0	2	1	1	0.200000	...	0	0	0	0	0	0	0	0	
1	0	3	1	1	0.226957	...	0	0	0	0	0	0	0	0	

Splitting the Data

To properly split the data, we will apply the following four steps:

1) set the number of train (total - (test+train) , test (50) and holdout (50) values
2) create new variable names containing the proper number of samples
3) create the target variables dataset based on its corresponding sample
4) drop the target variable column from the train, test and hold samples (NOTE: for simplicity, this step will be done after we prepare 3-D

In [230]:
```
# Splitting the Data: Step 1

number_of_test_data = 50
number_of_holdout_data = 50
number_of_training_data = len(dataset) - number_of_holdout_data - number_of_test_data
print ("total, train, test, holdout:", len(dataset), number_of_training_data, number_of_test_data, number_of_h
```

```
total, train, test, holdout: 731 631 50 50
```

In [231]:
```
# Splitting the Data: Step 2

datatrain = dataset[:number_of_training_data]
datatest = dataset[-(number_of_test_data+number_of_holdout_data):-number_of_holdout_data]
datahold = dataset[-number_of_holdout_data:]
```

In [232]:
```
# Splitting the Data: Step 3

y_train = datatrain['cntscl']
y_test = datatest['cntscl']
y_hold = datahold['cntscl']
```

In [233]:
```
# Splitting the Data: Step 4

datatrain = datatrain.drop(columns=['cntscl'])
datatest = datatest.drop(columns=['cntscl'])
datahold = datahold.drop(columns=['cntscl'])
```

Preparing 3-Dimensional Input for Sequential Model

Preparing input for a sequential model by using TimeSeriesGenerator. The following code iterates over the list of features and performs reshaping operations within the loop.

In [234]:

```
# Preparing Input for Sequential Model
n_input = 10
features = ['holiday', 'workingday', 'temp', 'atemp', 'hum', 'windspeed',
            'weekday_0', 'weekday_1', 'weekday_2', 'weekday_3', 'weekday_4', 'weekday_5', 'weekday_6',
            'weathersit_1', 'weathersit_2', 'weathersit_3']

input_seqs_train = []
input_seqs_test = []
input_seqs_hold = []

for feature in features:
    reshaped_feature_train = datatrain[feature].values.reshape((len(datatrain), 1))
    reshaped_feature_test = datatest[feature].values.reshape((len(datatest), 1))
    reshaped_feature_hold = datahold[feature].values.reshape((len(datahold), 1))

    input_seqs_train.append(reshaped_feature_train)
    input_seqs_test.append(reshaped_feature_test)
    input_seqs_hold.append(reshaped_feature_hold)

datatrain_feed = np.hstack(input_seqs_train)
datatest_feed = np.hstack(input_seqs_test)
datahold_feed = np.hstack(input_seqs_hold)

# Ensure y_train has the same length as datatrain
y_train = y_train[:len(datatrain)]

# Initialize TimeseriesGenerator for training, testing, and holdout sets
batch_size_train = 1  # Adjust this value as needed
batch_size_test = 1   # Adjust this value as needed
batch_size_hold = 1   # Adjust this value as needed

generator_train = TimeseriesGenerator(datatrain_feed, y_train, length=n_input, batch_size=batch_size_train)
generator_test = TimeseriesGenerator(datatest_feed, y_test, length=n_input, batch_size=batch_size_test)
generator_hold = TimeseriesGenerator(datahold_feed, y_hold, length=n_input, batch_size=batch_size_hold)

# Truncate y_train to match the length of generator_train
y_train = y_train[:len(generator_train)]
y_test = y_train[:len(generator_test)]
y_hold = y_train[:len(generator_hold)]
```

```
Length of y_train: 621
Length of y_test: 40
Length of y_hold: 40
Length of generator_train: 621
Length of generator_test: 40
Length of generator_hold: 40
```

In [235]:

```
# Print Lengths
print("Length of y_train:", len(y_train))
print("Length of y_test:", len(y_test))
print("Length of y_hold:", len(y_hold))

print("Length of generator_train:", len(generator_train))
print("Length of generator_test:", len(generator_test))
print("Length of generator_hold:", len(generator_hold))
```

```
Length of y_train: 621
Length of y_test: 40
Length of y_hold: 40
Length of generator_train: 621
Length of generator_test: 40
Length of generator_hold: 40
```

Part IV: Building and Training Simple RNN model

We will now create a small RNN with 4 nodes. Number of total parameters in the model is 93. Number of timesteps in one batch is 10. Activation function is relu both for RNN and Output layer. Optimizer is adam. Loss function is mean squared error. Learning rate is 0.0001. Number of epocs is 3,000.

Creating the Model

In [32]:
```
model = Sequential()

model.add(SimpleRNN(4, activation='relu', input_shape=(n_input, n_features), return_sequences = False))
model.add(Dense(1, activation='relu'))

adam = Adam(lr=0.0001)
model.compile(optimizer=adam, loss='mse')
```

```
C:\Users\mulli\anaconda3\lib\site-packages\keras\optimizers\optimizer_v2\adam.py:117: UserWarning: The `lr` a
rgument is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

In [33]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 4)                 84

 dense (Dense)               (None, 1)                 5

=================================================================
Total params: 89
Trainable params: 89
Non-trainable params: 0
_________________________________________________________________
```

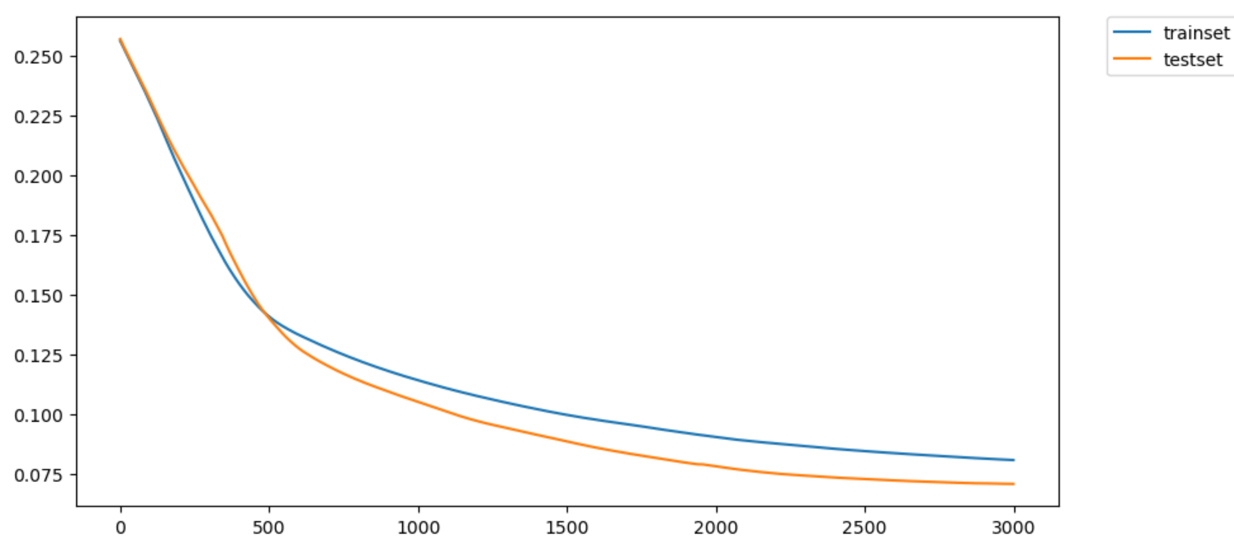
Training the Model

In [35]:
```
score = model.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_test)
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\1338044232.py:1: UserWarning: `Model.fit_generator` is depr
ecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  score = model.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_test)
```

Plot of Training and Test Loss Functions

In [36]:
```
losses = score.history['loss']
val_losses = score.history['val_loss']
plt.figure(figsize=(10,5))
plt.plot(losses, label="trainset")
plt.plot(val_losses, label="testset")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



Predictions for Test Data

In [57]:

```
# Predicting for Test Data
df_result = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_test)):
    x, y = generator_test[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model.predict(x_input, verbose=2)

    # Inverse transform the scaled target values to their original scale
    actual = scaler.inverse_transform(y.reshape(-1, 1))[0][0]
    prediction = scaler.inverse_transform(yhat.reshape(-1, 1))[0][0]

    df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)

# Display the results
print(df_result)
```

```
27  5923.154798  5416.117676
28  2116.803704  5651.955566
29  6059.796630  3977.350342
30  4965.051879  3103.239502
31   751.609809  4303.803711
32  3521.631447  3060.876709
33  5831.070164  3863.071777
34  2830.350504  4241.522461
35   701.774752  4152.302246
36  3777.680203  4223.648926
37   811.816748  3457.753418
38  4745.583842  4638.375977
39  1451.312076  4290.109375

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2465833379.py:13: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2465833379.py:13: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)
```

Tabulating Actuals, Predictions and Differences

In [58]:

```
df_result['Diff'] = 100 * (df_result['Prediction'] - df_result['Actual']) / df_result['Actual']
```

In [59]:
```
df_result
```

Out[59]:

	Actual	Prediction	Diff
0	5747.700363	4689.526367	-18.410389
1	7062.688266	4303.365234	-39.069019
2	4344.591062	3180.356201	-26.797341
3	7140.891501	4986.168945	-30.174419
4	6131.928586	4217.518555	-31.220358
5	7912.312059	4231.909668	-46.514879
6	6691.261626	3993.219971	-40.321868
7	7985.740997	4787.211914	-40.053003
8	5015.049197	4337.256836	-13.515169
9	4202.067943	3722.940674	-11.402178
10	644.751835	4795.160156	643.721831
11	899.480194	4117.308105	357.743053
12	8219.107814	4263.191895	-48.130722
13	2335.952194	4753.149414	103.478026
14	7645.490248	3806.860596	-50.207763
15	7819.662325	5121.740234	-34.501772
16	4857.324591	3366.904785	-30.683966
17	4855.510698	4584.152832	-5.588658
18	5383.215834	4216.047852	-21.681612
19	3426.729103	4545.614258	32.651696
20	2208.090023	5034.458984	128.000622
21	6797.785140	4414.158691	-35.064751
22	2819.375201	5121.942383	81.669413
23	2875.522578	3539.791504	23.100807
24	5695.967169	4669.700684	-18.017423
25	2581.427003	3536.294678	36.989916
26	3242.468071	3420.623535	5.494440
27	5923.154798	5416.117676	-8.560254
28	2116.803704	5651.955566	167.004236
29	6059.796630	3977.350342	-34.364953
30	4965.051879	3103.239502	-37.498347
31	751.609809	4303.803711	472.611435
32	3521.631447	3060.876709	-13.083559
33	5831.070164	3863.071777	-33.750209
34	2830.350504	4241.522461	49.858558
35	701.774752	4152.302246	491.685898
36	3777.680203	4223.648926	11.805359
37	811.816748	3457.753418	325.927825
38	4745.583842	4638.375977	-2.259108
39	1451.312076	4290.109375	195.602128

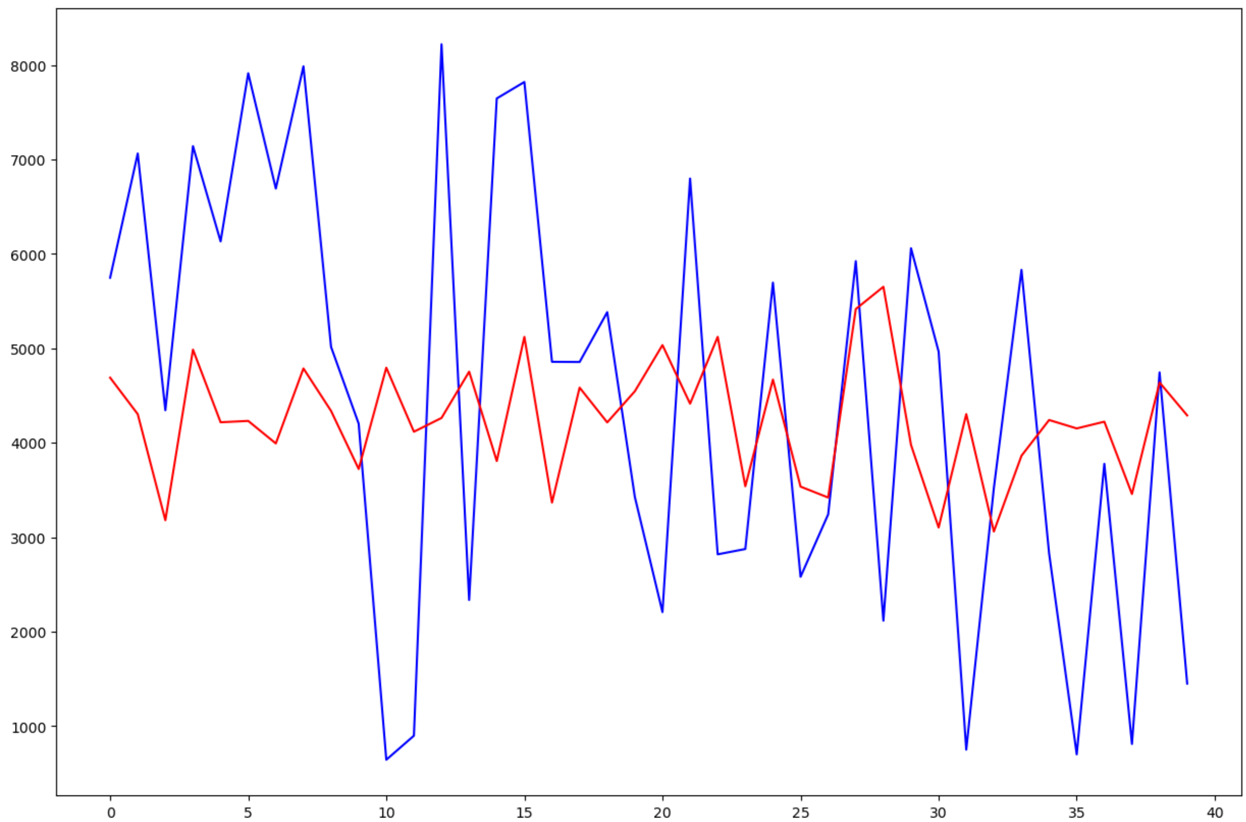
Calculating the Correctness for Test Data

In [60]:
```
# Calculating the Correctness for Test Data
mean_actual = df_result['Actual'].mean()
mae = (df_result['Actual'] - df_result['Prediction']).abs().mean()
mae_percentage = 100 * mae / mean_actual
correctness = 100 - mae_percentage

# Print the statistics
print("Mean Actual: ", mean_actual)
print("Mean Absolute Error (MAE):", mae)
print("MAE/Mean ratio: ", mae_percentage,"%")
print("Correctness: ", correctness,"%")
```

```
Mean Actual:  4450.743204438795
Mean Absolute Error (MAE): 1982.8403745712044
MAE/Mean ratio:  44.55077014089887 %
Correctness:  55.44922985910113 %
```

Plot of Actuals and Predictions for Test Data

In [50]:
```
plt.figure(figsize=(15,10))
plt.plot(df_result['Actual'], color='blue')
plt.plot(df_result['Prediction'], color='red')
plt.show()
```



Observations

Model 1 (Simple RNN) Train Data Statistics:

- MAE: 1982.84
- MAE/Mean Ratio: 44.55%
- Correctness: 55.45%

The red line (prediction of test data) on the predictions plot actually appears less volatile than the blue line (actual), which could mean that we are losing the seasonality component of the the time series. We can see that with a correctness score of just 55%, this model is probably not functioning as optimally as it could.

Predictions for Hold-Out Data

In [61]:

```
# Predictions for Hold-Out Data
df_result = pd.DataFrame({'Actual' : [], 'Prediction' : []})

for i in range(len(generator_hold)):
    x, y = generator_hold[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model.predict(x_input, verbose=2)

    # Reshape y to 2D array if it's 1D
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Reshape yhat to 2D array if it's 1D
    if len(yhat.shape) == 1:
        yhat = yhat.reshape(-1, 1)

    # Inverse transform the scaled target values to their original scale
    actual = scaler.inverse_transform(y)[0][0]
    prediction = scaler.inverse_transform(yhat)[0][0]

    df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)
```

```
1/1 - 0s - 24ms/epoch - 24ms/step
1/1 - 0s - 22ms/epoch - 22ms/step
1/1 - 0s - 25ms/epoch - 25ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2791513956.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2791513956.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2791513956.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result = df_result.append({'Actual': actual, 'Prediction': prediction}, ignore_index=True)

1/1 - 0s - 29ms/epoch - 29ms/step
1/1 - 0s - 25ms/epoch - 25ms/step
1/1 - 0s - 34ms/epoch - 34ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2791513956.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
```

Tabulating Actuals, Predictions and Differences for Hold-Out Data

In [62]:
```
df_result['Diff'] = 100 * (df_result['Prediction'] - df_result['Actual']) / df_result['Actual']
df_result
```

Out[62]:

	Actual	Prediction	Diff
0	3031.632502	4603.979004	51.864680
1	557.736274	3035.659668	444.282272
2	941.392674	4522.917969	380.449667
3	4873.155902	3729.394531	-23.470650
4	2797.711835	4127.327637	47.525116
5	8543.326365	4837.718262	-43.374301
6	2101.122616	4875.093750	132.023287
7	976.422904	4535.048340	364.455342
8	3318.467790	4061.083984	22.378285
9	2315.290270	5315.419434	129.578965
10	7114.650952	4035.322021	-43.281518
11	6694.469069	4072.714355	-39.162997
12	5355.566661	4797.842773	-10.413910
13	8004.981553	4346.573242	-45.701646
14	1048.273425	4807.707031	358.631013
15	3397.319630	4042.985107	19.005144
16	7475.350299	4753.871094	-36.406042
17	4937.293928	3975.796387	-19.474181
18	2560.269101	3992.083496	55.924371
19	4120.945793	4509.436523	9.427223
20	1559.693308	4171.667969	167.467197
21	639.915284	3654.338379	471.065963
22	520.950104	3710.507568	612.257765
23	1944.917049	4925.457520	153.247691
24	1327.880375	4360.150879	228.354192
25	6308.385150	4325.540527	-31.431889
26	3931.703909	4867.803711	23.809011
27	8626.042641	4031.779053	-53.260386
28	4670.567860	5111.006836	9.430095
29	7528.444553	3531.629150	-53.089524
30	4054.759589	3847.758057	-5.105149
31	2323.295978	3668.436768	57.897952
32	4694.935671	4141.073242	-11.797018
33	6978.060568	4658.645020	-33.238685
34	8141.866331	2894.030029	-64.454955
35	6392.913456	4473.832031	-30.018886
36	3247.792461	3250.342529	0.078517
37	3492.756772	4185.858398	19.843971
38	5273.871623	3673.551514	-30.344313
39	6117.423293	3867.126709	-36.785040

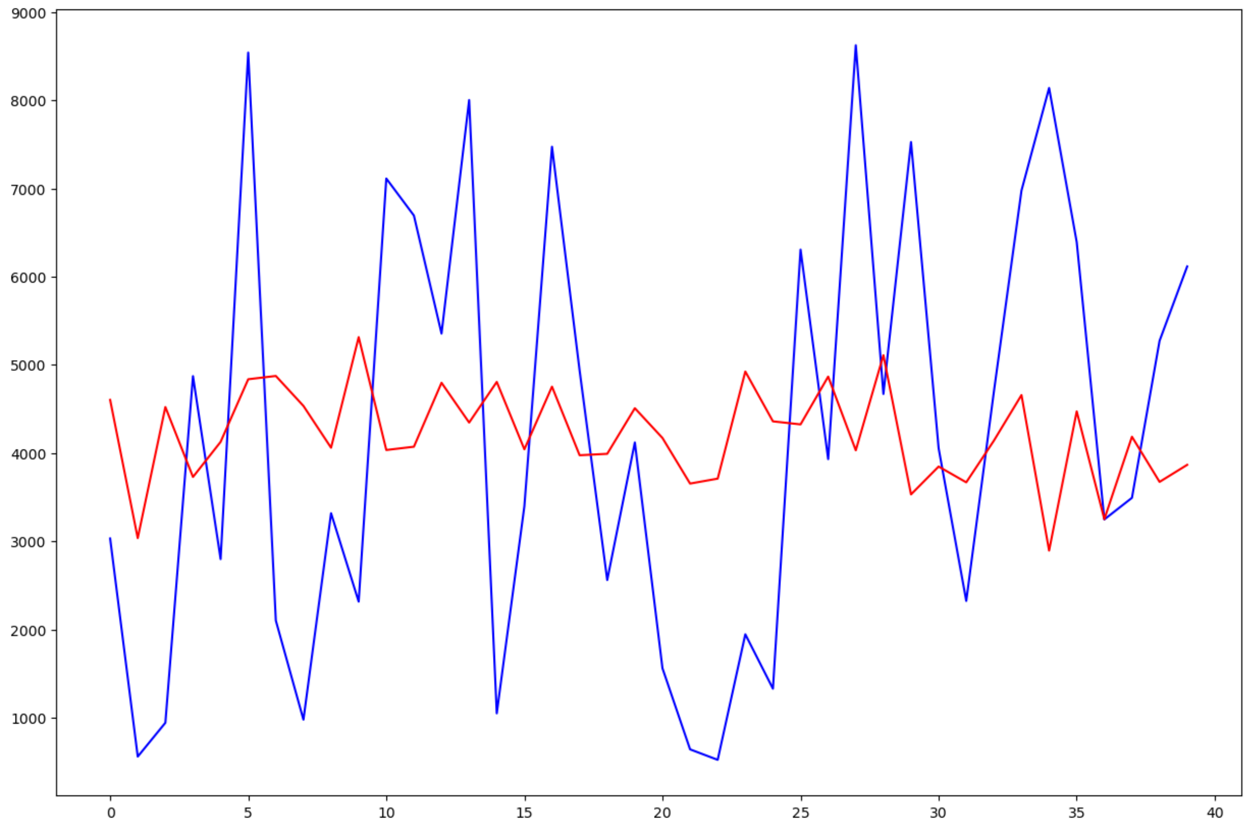
Calculating the Correctness for Hold-Out Data

In [63]:
```
mean = df_result['Actual'].mean()
mae = (df_result['Actual'] - df_result['Prediction']).abs().mean()

print("mean: ", mean)
print("mae:", mae)
print("mae/mean ratio: ", 100*mae/mean,"%")
print("correctness: ", 100 - 100*mae/mean,"%")
```

```
mean:  4198.538887975436
mae: 2165.738854755758
mae/mean ratio:  51.58315577255715 %
correctness:  48.41684422744285 %
```

Plot of Actuals and Predictions for Hold-Out Data

In [111]:
```
plt.figure(figsize=(15,10))
plt.plot(df_result['Actual'], color='blue')
plt.plot(df_result['Prediction'], color='red')
plt.show()
```



Observations

Model 1 (Simple RNN) Holdout Data Statistics:

- Mean Absolute Error (MAE): 2165.74
- MAE/Mean ratio: 51.58%
- Correctness: 48.42%

The red line (prediction of test data) on the predictions plot actually appears less volatile than the blue line (actual), which could mean that we are losing the seasonality component of the the time series. We can see that with a correctness score of just 48%, this model is probably not functioning as optimally as it could.

Model 1 Overall Observations

Mean Absolute Error (MAE):

Value: 1982.84

Explanation: The Mean Absolute Error (MAE) measures the average absolute difference between the actual and predicted values. In the context of Model 1, an MAE of 1982.84 indicates that, on average, the predictions made by the Simple RNN model deviate from the actual values by approximately 1982.84 units. Lower values of MAE indicate better predictive performance, as they signify smaller errors between predictions and actual observations.

MAE/Mean Ratio:

Value: 44.55%

Explanation: The MAE/Mean Ratio, also known as the Mean Absolute Percentage Error (MAPE), expresses the MAE as a percentage of the mean of the actual values. In Model 1, a ratio of 44.55% suggests that the average absolute error in predictions, relative to the mean of the actual values, is approximately 44.55%. This metric provides a normalized measure of prediction accuracy, allowing for comparisons across different datasets and models. Lower values indicate higher accuracy, as they represent smaller relative errors.

Correctness:

Value: 55.45%

Explanation: The Correctness metric represents the proportion of predictions that are considered correct, typically based on a predefined threshold or criterion. In Model 1, a correctness of 55.45% indicates that approximately 55.45% of the predictions made by the Simple RNN model align with the actual observations. This metric provides a binary assessment of prediction accuracy, where higher values indicate a higher proportion of accurate predictions.

In summary, these evaluation metrics provide insights into the predictive performance of Model 1 based on its ability to minimize absolute errors, maintain low relative errors, and achieve correctness in its predictions.

Part V: Building and Training a GRU (Gated Recurrent Unit) Model

Key features of GRU include:

Gating Mechanism: GRU incorporates gating mechanisms similar to Long Short-Term Memory (LSTM) networks, which allow it to selectively update and forget information over time. However, GRU has a simpler architecture compared to LSTM, with two gates: the update gate and the reset gate.

Update Gate: The update gate determines how much of the past information should be carried forward to the current time step. It takes into account the current input and the previous hidden state and outputs a value between 0 and 1, indicating the proportion of information to retain.

Reset Gate: The reset gate controls how much of the past information should be forgotten when computing the current hidden state. It helps the model adaptively reset its memory based on the current input.

Efficiency: GRU has fewer parameters compared to LSTM, making it computationally more efficient and faster to train. This can be advantageous, especially when dealing with large datasets or complex models.

Overall, GRU networks have shown promising performance in various sequential data tasks and have become a popular choice alongside LSTM networks in many deep learning applications.

Creating the Model

In [66]:

```
# Create the GRU model
model_gru = Sequential()

# Add a GRU layer with 4 units and 'relu' activation function
model_gru.add(GRU(4, activation='relu', input_shape=(n_input, n_features)))

# Add a Dense output layer
model_gru.add(Dense(1, activation='relu'))

# Compile the model
adam = Adam(lr=0.0001)
model_gru.compile(optimizer=adam, loss='mse')
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 4)                 264

 dense_1 (Dense)             (None, 1)                 5

=================================================================
Total params: 269
Trainable params: 269
Non-trainable params: 0
_________________________________________________________________

C:\Users\mulli\anaconda3\lib\site-packages\keras\optimizers\optimizer_v2\adam.py:117: UserWarning: The `lr` a
rgument is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

In [68]:

```
# Print the summary of the model
model_gru.summary()
```


```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 4)                 264

 dense_1 (Dense)             (None, 1)                 5

=================================================================
Total params: 269
Trainable params: 269
Non-trainable params: 0
_________________________________________________________________
```


Training the Model

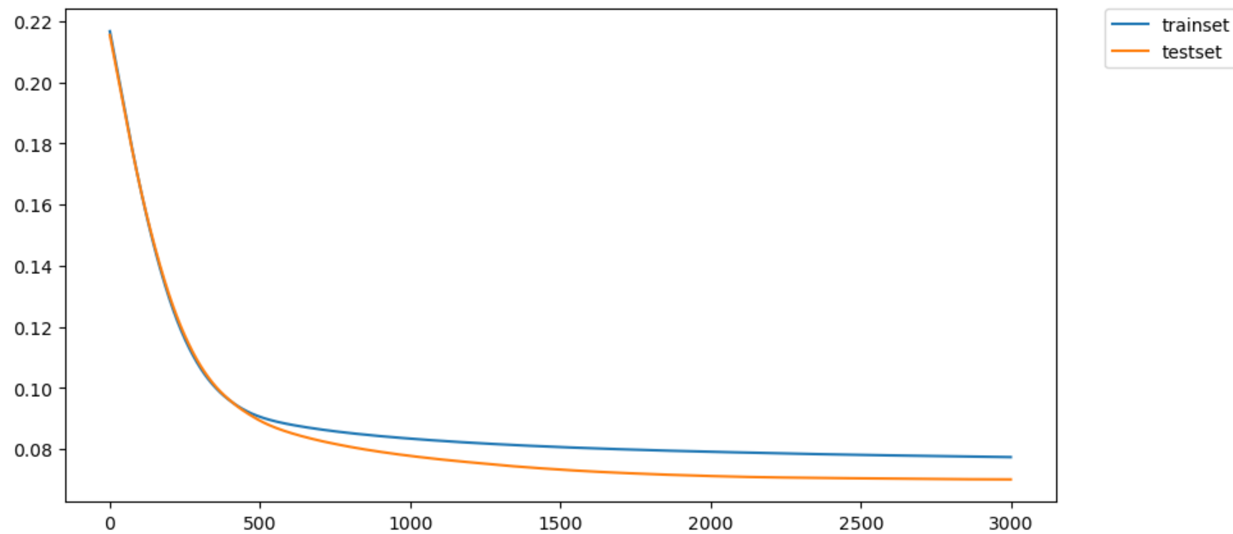
In [69]:

```
# Training the GRU Model
score_gru = model_gru.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_test)
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2218321493.py:2: UserWarning: `Model.fit_generator` is depr
ecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  score_gru = model_gru.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_tes
t)
```

Plot of Training and Test Loss Functions

In [70]:
```
# Plotting Training and Test Loss Functions
losses_gru = score_gru.history['loss']
val_losses_gru = score_gru.history['val_loss']
plt.figure(figsize=(10,5))
plt.plot(losses_gru, label="trainset")
plt.plot(val_losses_gru, label="testset")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



Predictions for Test Data

In [71]:
```
# Predictions for Test Data
df_result_gru = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_test)):
    x, y = generator_test[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model_gru.predict(x_input, verbose=2)

    # Inverse transform the scaled target values to their original scale
    actual_gru = scaler.inverse_transform(y.reshape(-1, 1))[0][0]
    prediction_gru = scaler.inverse_transform(yhat.reshape(-1, 1))[0][0]

    df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=Tr

# Display the results
print(df_result_gru)
```

```
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=
True)

1/1 - 0s - 44ms/epoch - 44ms/step
1/1 - 0s - 41ms/epoch - 41ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2512105805.py:13: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=
True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2512105805.py:13: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=
True)

1/1 - 0s - 32ms/epoch - 32ms/step
1/1 - 0s - 39ms/epoch - 39ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2512105805.py:13: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=
```

Tabulating Actuals, Predictions and Differences

In [72]:

```
# Tabulating Actuals, Predictions and Differences
df_result_gru['Diff'] = 100 * (df_result_gru['Prediction'] - df_result_gru['Actual']) / df_result_gru['Actual'
```

```
         Actual   Prediction        Diff
0   5747.700363  4398.889648  -23.466963
1   7062.688266  4261.813477  -39.657347
2   4344.591062  4303.301270   -0.950372
3   7140.891501  4758.508301  -33.362546
4   6131.928586  4084.322754  -33.392526
5   7912.312059  3996.922363  -49.484773
6   6691.261626  5295.481445  -20.859746
7   7985.740997  4947.979492  -38.039820
8   5015.049197  4207.316895  -16.106169
9   4202.067943  4197.134766   -0.117399
10   644.751835  4703.789551  629.550393
11   899.480194  3985.646484  343.105530
12  8219.107814  4124.873535  -49.813610
13  2335.952194  5141.087891  120.085321
14  7645.490248  4466.256836  -41.583120
15  7819.662325  3872.254395  -50.480542
16  4857.324591  4110.826660  -15.368500
17  4855.510698  4746.234375   -2.250563
18  5383.215834  3985.520264  -25.963952
19  3426.729103  3999.934326   16.727474
20  2208.090023  4481.339355  102.950935
21  6797.785140  4044.782227  -40.498528
22  2819.375201  3627.550049   28.665034
23  2875.522578  3878.852539   34.892091
24  5695.967169  4603.065918  -19.187281
25  2581.427003  4464.817383   72.959273
26  3242.468071  4331.248535   33.578757
27  5923.154798  5070.269043  -14.399181
28  2116.803704  4828.218750  128.090056
29  6059.796630  4425.070801  -26.976579
30  4965.051879  4421.656738  -10.944400
31   751.609809  4988.630371  563.726087
32  3521.631447  4725.897949   34.196267
33  5831.070164  4560.845215  -21.783736
34  2830.350504  4871.482422   72.115871
35   701.774752  4229.592773  502.699479
36  3777.680203  4114.029297    8.903588
37   811.816748  4108.119629  406.040266
38  4745.583842  4706.172363   -0.830487
39  1451.312076  3919.425781  170.060854
```

In [73]:
```
print(df_result_gru)
```

```
         Actual   Prediction        Diff
0   5747.700363  4398.889648  -23.466963
1   7062.688266  4261.813477  -39.657347
2   4344.591062  4303.301270   -0.950372
3   7140.891501  4758.508301  -33.362546
4   6131.928586  4084.322754  -33.392526
5   7912.312059  3996.922363  -49.484773
6   6691.261626  5295.481445  -20.859746
7   7985.740997  4947.979492  -38.039820
8   5015.049197  4207.316895  -16.106169
9   4202.067943  4197.134766   -0.117399
10   644.751835  4703.789551  629.550393
11   899.480194  3985.646484  343.105530
12  8219.107814  4124.873535  -49.813610
13  2335.952194  5141.087891  120.085321
14  7645.490248  4466.256836  -41.583120
15  7819.662325  3872.254395  -50.480542
16  4857.324591  4110.826660  -15.368500
17  4855.510698  4746.234375   -2.250563
18  5383.215834  3985.520264  -25.963952
19  3426.729103  3999.934326   16.727474
20  2208.090023  4481.339355  102.950935
21  6797.785140  4044.782227  -40.498528
22  2819.375201  3627.550049   28.665034
23  2875.522578  3878.852539   34.892091
24  5695.967169  4603.065918  -19.187281
25  2581.427003  4464.817383   72.959273
26  3242.468071  4331.248535   33.578757
27  5923.154798  5070.269043  -14.399181
28  2116.803704  4828.218750  128.090056
29  6059.796630  4425.070801  -26.976579
30  4965.051879  4421.656738  -10.944400
31   751.609809  4988.630371  563.726087
32  3521.631447  4725.897949   34.196267
33  5831.070164  4560.845215  -21.783736
34  2830.350504  4871.482422   72.115871
35   701.774752  4229.592773  502.699479
36  3777.680203  4114.029297    8.903588
37   811.816748  4108.119629  406.040266
38  4745.583842  4706.172363   -0.830487
39  1451.312076  3919.425781  170.060854
```

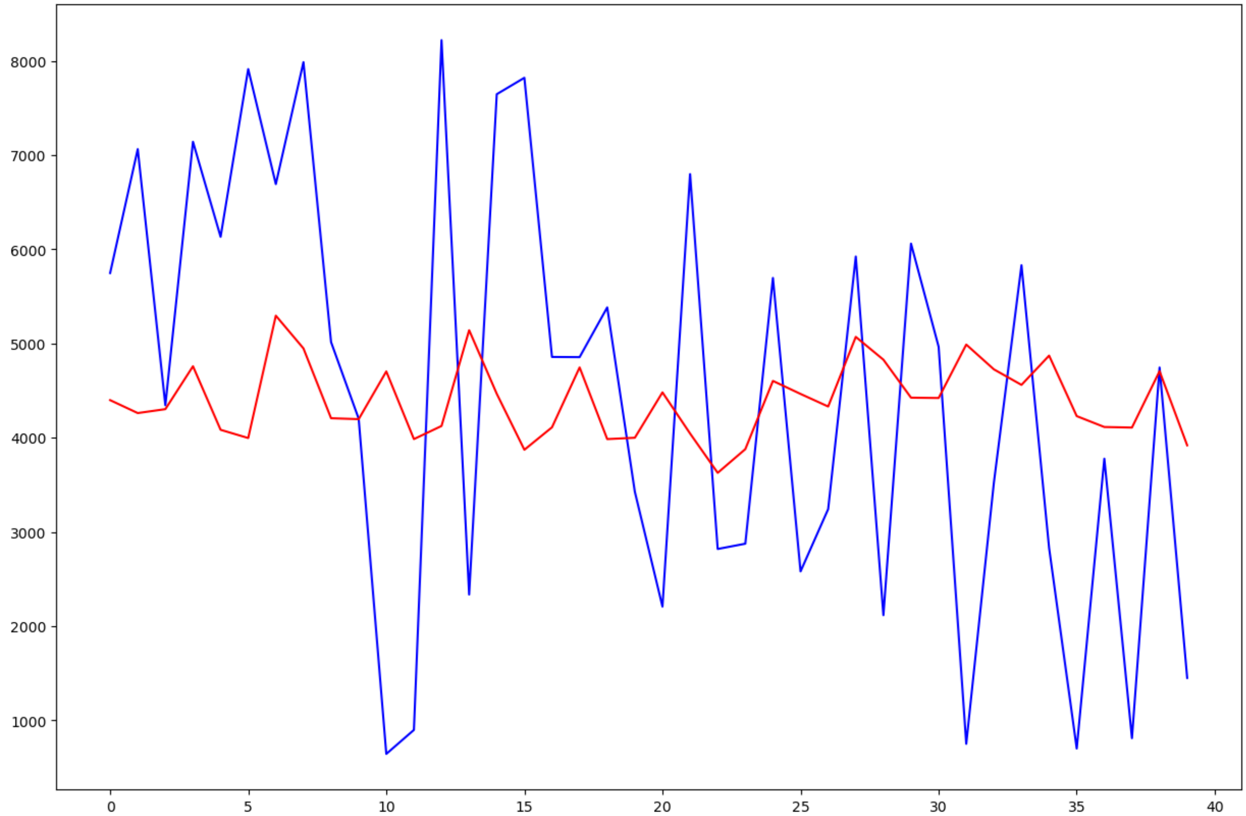
Calculating the Correctness for Test Data

In [74]:
```
# Calculating the Correctness for Test Data
mean_actual_gru = df_result_gru['Actual'].mean()
mae_gru = (df_result_gru['Actual'] - df_result_gru['Prediction']).abs().mean()
mae_percentage_gru = 100 * mae_gru / mean_actual_gru
correctness_gru = 100 - mae_percentage_gru

# Print the statistics
print("Mean Actual: ", mean_actual_gru)
print("Mean Absolute Error (MAE):", mae_gru)
print("MAE/Mean ratio: ", mae_percentage_gru,"%")
print("Correctness: ", correctness_gru,"%")
```

```
Mean Actual:  4450.743204438795
Mean Absolute Error (MAE): 1921.1585397935348
MAE/Mean ratio:  43.16489295265414 %
Correctness:  56.83510704734586 %
```

Plot of Actuals and Predictions for Test Data

In [75]:
```
# Plot of Actuals and Predictions for Test Data
plt.figure(figsize=(15,10))
plt.plot(df_result_gru['Actual'], color='blue')
plt.plot(df_result_gru['Prediction'], color='red')
plt.show()
```



Predictions for Hold-Out Data

In [76]:

```
# Predictions for Hold-Out Data
df_result_hold = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_hold)):
    x, y = generator_hold[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model_gru.predict(x_input, verbose=2)

    # Reshape y to 2D array if it's 1D
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Reshape yhat to 2D array if it's 1D
    if len(yhat.shape) == 1:
        yhat = yhat.reshape(-1, 1)

    # Inverse transform the scaled target values to their original scale
    actual_hold = scaler.inverse_transform(y)[0][0]
    prediction_hold = scaler.inverse_transform(yhat)[0][0]

    df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_inde

# Display the results
print(df_result_hold)
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2256054054.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_in
dex=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2256054054.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_in
dex=True)

1/1 - 0s - 43ms/epoch - 43ms/step
1/1 - 0s - 33ms/epoch - 33ms/step
1/1 - 0s - 27ms/epoch - 27ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2256054054.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_in
dex=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\2256054054.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
```

Tabulating Actuals, Predictions and Differences for Hold-Out Data

In [77]:
```
# Tabulating Actuals, Predictions and Differences for Hold-Out Data
df_result_hold['Diff'] = 100 * (df_result_hold['Prediction'] - df_result_hold['Actual']) / df_result_hold['Act
print(df_result_hold)
```

```
         Actual   Prediction        Diff
0   3031.632502  3944.280273   30.104169
1    557.736274  3947.070312  607.694746
2    941.392674  4550.256836  383.353755
3   4873.155902  3867.883301  -20.628780
4   2797.711835  4040.079346   44.406557
5   8543.326365  4403.373047  -48.458330
6   2101.122616  4379.104492  108.417370
7    976.422904  3669.067627  275.766239
8   3318.467790  3871.342285   16.660535
9   2315.290270  4525.254395   95.450845
10  7114.650952  4401.266113  -38.137990
11  6694.469069  4253.926758  -36.456100
12  5355.566661  4659.327148  -13.000296
13  8004.981553  4184.001465  -47.732528
14  1048.273425  3670.655518  250.162031
15  3397.319630  3949.786621   16.261849
16  7475.350299  4768.507324  -36.210249
17  4937.293928  4525.319824   -8.344128
18  2560.269101  4384.117676   71.236597
19  4120.945793  5321.583496   29.135004
20  1559.693308  4970.196289  218.664975
21   639.915284  4458.052734  596.662956
22   520.950104  4349.240234  734.866948
23  1944.917049  4920.909180  153.013833
24  1327.880375  4104.472168  209.099543
25  6308.385150  4116.879883  -34.739560
26  3931.703909  5050.313965   28.451025
27  8626.042641  4454.785156  -48.356560
28  4670.567860  3809.043213  -18.445822
29  7528.444553  4090.893799  -45.660836
30  4054.759589  4959.675293   22.317370
31  2323.295978  4005.532227   72.407316
32  4694.935671  4121.321777  -12.217716
33  6978.060568  5031.952148  -27.888959
34  8141.866331  5222.166016  -35.860332
35  6392.913456  4938.998047  -22.742611
36  3247.792461  5110.996094   57.368310
37  3492.756772  5659.572754   62.037414
38  5273.871623  5064.203125   -3.975609
39  6117.423293  4965.919922  -18.823340
```

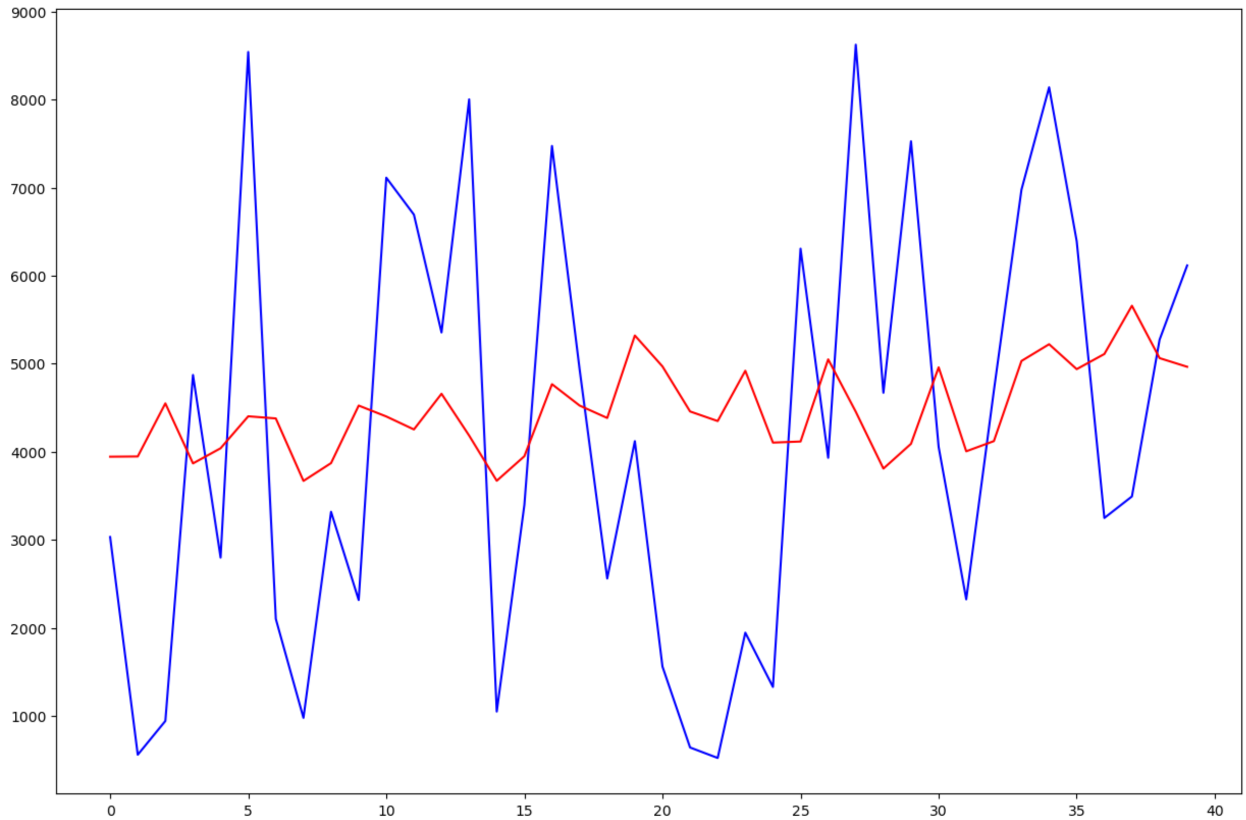
Calculating the Correctness for Hold-Out Data

In [78]:
```
# Calculating the Correctness for Hold-Out Data
mean_hold = df_result_hold['Actual'].mean()
mae_hold = (df_result_hold['Actual'] - df_result_hold['Prediction']).abs().mean()
mae_percentage_hold = 100 * mae_hold / mean_hold
correctness_hold = 100 - mae_percentage_hold

# Print the statistics
print("Mean Actual: ", mean_hold)
print("Mean Absolute Error (MAE):", mae_hold)
print("MAE/Mean ratio: ", mae_percentage_hold,"%")
print("Correctness: ", correctness_hold,"%")
```

```
Mean Actual:  4198.538887975436
Mean Absolute Error (MAE): 2112.071199426272
MAE/Mean ratio:  50.3049097740935 %
Correctness:  49.6950902259065 %
```

Plot of Actuals and Predictions for Hold-Out Data

In [79]:
```
# Plot of Actuals and Predictions for Hold-Out Data
plt.figure(figsize=(15,10))
plt.plot(df_result_hold['Actual'], color='blue')
plt.plot(df_result_hold['Prediction'], color='red')
plt.show()
```



Model 2 Overeall Observations

Mean Absolute Error (MAE):

Value: 1921.16

Explanation: The MAE measures the average absolute difference between the actual and predicted values. In the context of Model 2 (GRU), an MAE of 1921.16 indicates that, on average, the predictions made by the GRU model deviate from the actual values by approximately 1921.16 units. Lower values of MAE indicate better predictive performance, as they signify smaller errors between predictions and actual observations.

MAE/Mean Ratio:

Value: 43.16%

Explanation: The MAE/Mean Ratio, or Mean Absolute Percentage Error (MAPE), expresses the MAE as a percentage of the mean of the actual values. In Model 2 (GRU), a ratio of 43.16% suggests that the average absolute error in predictions, relative to the mean of the actual values, is approximately 43.16%. Lower values indicate higher accuracy, as they represent smaller relative errors.

Correctness:

Value: 55.45%

Explanation: The Correctness metric represents the proportion of predictions that are considered correct, typically based on a predefined threshold or criterion. In Model 1, a correctness of 55.45% indicates that approximately 55.45% of the predictions made by the Simple RNN model align with the actual observations. This metric provides a binary assessment of prediction accuracy, where higher values indicate a higher proportion of accurate predictions.

In summary, these evaluation metrics provide insights into the predictive performance of Model 1 based on its ability to minimize absolute errors, maintain low relative errors, and achieve correctness in its predictions.

Part VI: Building and Training a LSTM (Long Short-Term Memory) Model

LSTM stands for Long Short-Term Memory, which is a type of recurrent neural network (RNN) architecture designed to overcome the limitations of traditional RNNs in capturing long-term dependencies in sequential data.

Traditional RNNs suffer from the vanishing gradient problem, which occurs when gradients diminish exponentially as they are propagated back through time during training. As a result, traditional RNNs struggle to learn from long sequences of data, making them ineffective for tasks involving long-term dependencies.

LSTM networks address this issue by introducing a more complex recurrent unit called the LSTM cell. The key innovation of LSTM cells is their ability to maintain and update a memory state over time, allowing them to selectively remember or forget information from previous time steps.

An LSTM cell consists of several gates, including:

Forget Gate: Controls which information from the previous memory state should be discarded or forgotten. Input Gate: Determines which new information from the current input should be stored in the memory state. Output Gate: Regulates how much of the memory state should be revealed or used to compute the output at the current time step. By selectively updating and passing information through these gates, LSTM networks can effectively capture long-range dependencies in sequential data and mitigate the vanishing gradient problem.

LSTM networks have been widely used for various tasks, including time series forecasting, natural language processing, speech recognition, and more, where capturing temporal dependencies is essential.

In summary, LSTM networks are a type of RNN architecture designed to learn and remember long-term dependencies in sequential data by incorporating memory cells with gating mechanisms.

Creating the Model

In [85]:
```
# Define the LSTM model
model_lstm = Sequential()
model_lstm.add(LSTM(4, activation='relu', input_shape=(n_input, n_features)))
model_lstm.add(Dense(1, activation='relu'))

# Compile the model
adam = Adam(lr=0.0001)
model_lstm.compile(optimizer=adam, loss='mse')
```

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the legacy optimizer, e.g.,tf.ker
as.optimizers.legacy.Adam.
```

In [81]:
```
# Display the model summary
print(model_lstm.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 4)                 336

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 341
Trainable params: 341
Non-trainable params: 0
_________________________________________________________________
None
```

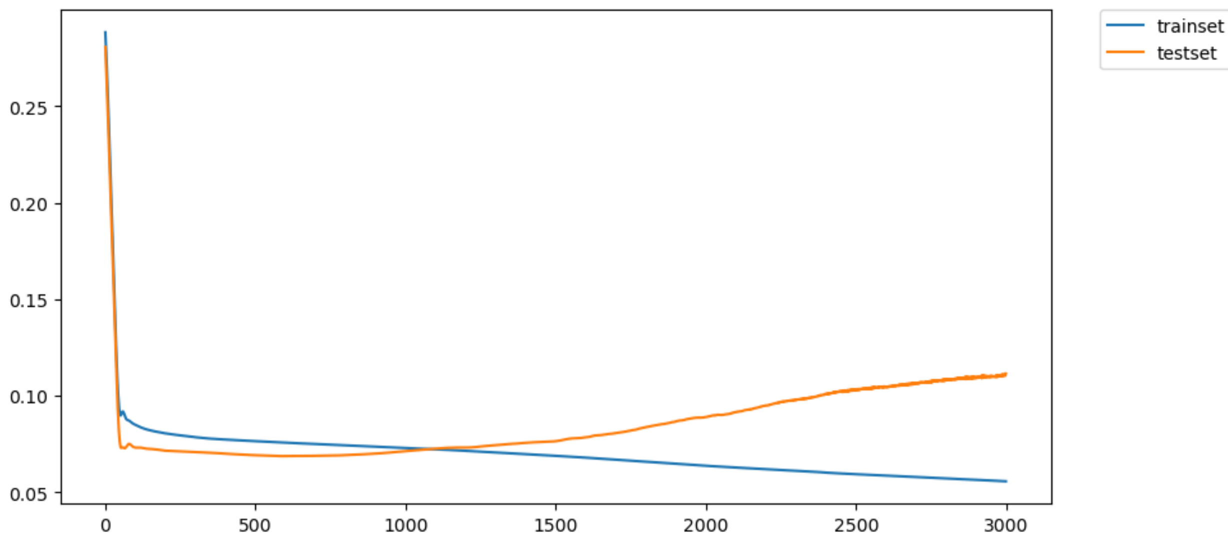
Training the Model

In [86]:
```
# Train the LSTM model
score_lstm = model_lstm.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_test)
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\1272318803.py:2: UserWarning: `Model.fit_generator` is depr
ecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  score_lstm = model_lstm.fit_generator(generator_train, epochs=3000, verbose=0, validation_data=generator_te
st)
```

Plot of Training and Test Loss Functions

In [87]:
```
# Plot the training and test loss functions
losses_lstm = score_lstm.history['loss']
val_losses_lstm = score_lstm.history['val_loss']
plt.figure(figsize=(10,5))
plt.plot(losses_lstm, label="trainset")
plt.plot(val_losses_lstm, label="testset")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



Predictions for Test Data

In [88]:

```
# Predictions for Test Data
df_result_test_lstm = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_test)):
    x, y = generator_test[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model_lstm.predict(x_input, verbose=2)

    # Reshape y to 2D array if it's 1D
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Reshape yhat to 2D array if it's 1D
    if len(yhat.shape) == 1:
        yhat = yhat.reshape(-1, 1)

    # Inverse transform the scaled target values to their original scale
    actual_test_lstm = scaler.inverse_transform(y)[0][0]
    prediction_test_lstm = scaler.inverse_transform(yhat)[0][0]

    df_result_test_lstm = df_result_test_lstm.append({'Actual': actual_test_lstm, 'Prediction': prediction_tes

# Display the results
print(df_result_test_lstm)
```

```
1/1 - 0s - 294ms/epoch - 294ms/step
1/1 - 0s - 34ms/epoch - 34ms/step
1/1 - 0s - 26ms/epoch - 26ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3088674518.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_test_lstm = df_result_test_lstm.append({'Actual': actual_test_lstm, 'Prediction': prediction_t
est_lstm}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3088674518.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_test_lstm = df_result_test_lstm.append({'Actual': actual_test_lstm, 'Prediction': prediction_t
est_lstm}, ignore_index=True)

1/1 - 0s - 40ms/epoch - 40ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3088674518.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_test_lstm = df_result_test_lstm.append({'Actual': actual_test_lstm, 'Prediction': prediction_t
est_lstm}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3088674518.py:21: FutureWarning: The frame.append method
```

Tabulating Actuals, Predictions and Differences

In [89]:

```
# Tabulating Actuals, Predictions and Differences for Test Data
df_result_test_lstm['Diff'] = 100 * (df_result_test_lstm['Prediction'] - df_result_test_lstm['Actual']) / df_r
```

In [90]:
```
print(df_result_test_lstm)
```

```
         Actual   Prediction        Diff
0   5747.700363  4525.071289  -21.271622
1   7062.688266  5113.988770  -27.591470
2   4344.591062  5021.780762   15.586961
3   7140.891501  4657.136230  -34.782145
4   6131.928586  3616.064209  -41.028925
5   7912.312059  3374.248291  -57.354459
6   6691.261626  2729.643311  -59.205850
7   7985.740997  2809.138916  -64.823065
8   5015.049197  3074.292725  -38.698653
9   4202.067943  3651.992920  -13.090579
10   644.751835  4549.296875  605.588821
11   899.480194  4479.096191  397.964960
12  8219.107814  4752.419434  -42.178403
13  2335.952194  5800.555664  148.316540
14  7645.490248  2025.304077  -73.509821
15  7819.662325  6276.296875  -19.736983
16  4857.324591  4585.538574   -5.595385
17  4855.510698  4986.946777    2.706947
18  5383.215834  3902.210205  -27.511541
19  3426.729103  1929.447998  -43.694178
20  2208.090023    22.000000  -99.003664
21  6797.785140  3143.271729  -53.760355
22  2819.375201  6911.675781  145.149201
23  2875.522578  4790.137695   66.583206
24  5695.967169  4396.843750  -22.807776
25  2581.427003  5283.712402  104.681844
26  3242.468071  5348.694824   64.957517
27  5923.154798  7112.480469   20.079260
28  2116.803704  5037.588867  137.980917
29  6059.796630  3105.083008  -48.759287
30  4965.051879  5003.362793    0.771612
31   751.609809  5561.558105  639.952838
32  3521.631447  5725.364258   62.577043
33  5831.070164  4400.153320  -24.539524
34  2830.350504  7059.516602  149.421992
35   701.774752  5540.770996  689.536953
36  3777.680203  3181.243896  -15.788428
37   811.816748  4089.165771  403.705520
38  4745.583842  4319.505371   -8.978420
39  1451.312076  4219.173828  190.714444
```

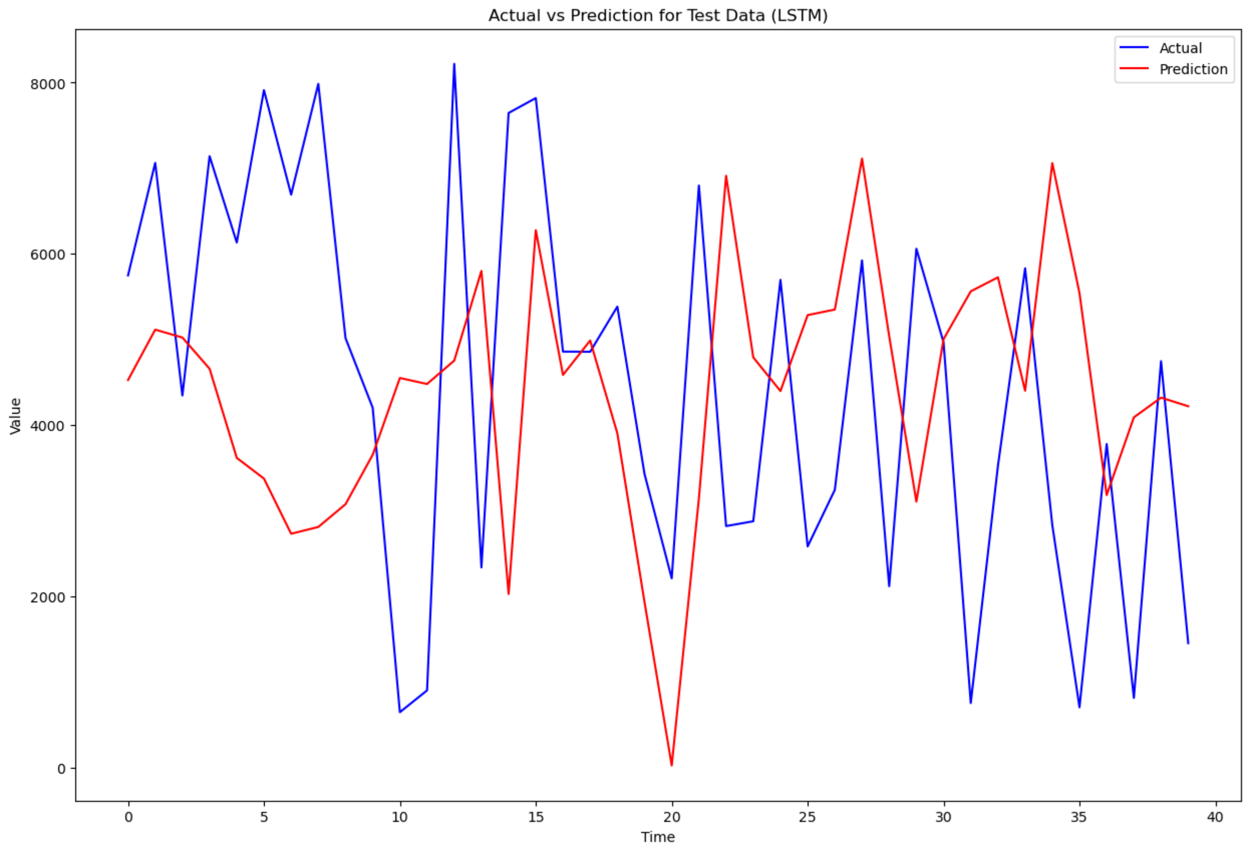
Calculating the Correctness for Test Data

In [91]:
```
# Calculating the Correctness for Test Data
mean_actual_lstm = df_result_test_lstm['Actual'].mean()
mae_lstm = (df_result_test_lstm['Actual'] - df_result_test_lstm['Prediction']).abs().mean()
mae_percentage_lstm = 100 * mae_lstm / mean_actual_lstm
correctness_lstm = 100 - mae_percentage_lstm

# Print the statistics
print("Mean Actual (LSTM): ", mean_actual_lstm)
print("Mean Absolute Error (MAE) for LSTM:", mae_lstm)
print("MAE/Mean ratio for LSTM: ", mae_percentage_lstm,"%")
print("Correctness for LSTM: ", correctness_lstm,"%")
```

```
Mean Actual (LSTM):  4450.743204438795
Mean Absolute Error (MAE) for LSTM: 2490.3635707845733
MAE/Mean ratio for LSTM:  55.953881327075784 %
Correctness for LSTM:  44.046118672924216 %
```

Plot of Actuals and Predictions for Test Data

In [92]:
```
plt.figure(figsize=(15,10))
plt.plot(df_result_test_lstm['Actual'], color='blue', label='Actual')
plt.plot(df_result_test_lstm['Prediction'], color='red', label='Prediction')
plt.title('Actual vs Prediction for Test Data (LSTM)')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```



Predictions for Hold-Out Data

In [93]:

```
# Predictions for Hold-Out Data
df_result_hold_lstm = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_hold)):
    x, y = generator_hold[i]
    x_input = array(x).reshape((1, n_input, n_features))
    yhat = model_lstm.predict(x_input, verbose=2)

    # Reshape y to 2D array if it's 1D
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Reshape yhat to 2D array if it's 1D
    if len(yhat.shape) == 1:
        yhat = yhat.reshape(-1, 1)

    # Inverse transform the scaled target values to their original scale
    actual_hold_lstm = scaler.inverse_transform(y)[0][0]
    prediction_hold_lstm = scaler.inverse_transform(yhat)[0][0]

    df_result_hold_lstm = df_result_hold_lstm.append({'Actual': actual_hold_lstm, 'Prediction': prediction_hol

# Display the results
print(df_result_hold_lstm)
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3272019864.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold_lstm = df_result_hold_lstm.append({'Actual': actual_hold_lstm, 'Prediction': prediction_h
old_lstm}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3272019864.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold_lstm = df_result_hold_lstm.append({'Actual': actual_hold_lstm, 'Prediction': prediction_h
old_lstm}, ignore_index=True)

1/1 - 0s - 24ms/epoch - 24ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3272019864.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold_lstm = df_result_hold_lstm.append({'Actual': actual_hold_lstm, 'Prediction': prediction_h
old_lstm}, ignore_index=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3272019864.py:21: FutureWarning: The frame.append method
is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold_lstm = df_result_hold_lstm.append({'Actual': actual_hold_lstm, 'Prediction': prediction_h
old_lstm}, ignore_index=True)
```

Tabulating Actuals, Predictions and Differences for Hold-Out Data

In [94]:
```
# Tabulating Actuals, Predictions and Differences for Hold-Out Data
df_result_hold_lstm['Diff'] = 100 * (df_result_hold_lstm['Prediction'] - df_result_hold_lstm['Actual']) / df_r
print(df_result_hold_lstm)
```

```
         Actual   Prediction        Diff
0   3031.632502  6740.432129  122.336715
1    557.736274  5720.306641  925.629299
2    941.392674  5479.397949  482.052325
3   4873.155902  5363.096680   10.053870
4   2797.711835  4793.604004   71.340162
5   8543.326365  3078.280273  -63.968598
6   2101.122616  8753.571289  316.614015
7    976.422904  9061.508789  828.031159
8   3318.467790  6467.996582   94.909126
9   2315.290270  4949.847168  113.789486
10  7114.650952  3546.347412  -50.154302
11  6694.469069  4265.100586  -36.289188
12  5355.566661  6078.669434   13.501891
13  8004.981553  5202.544434  -35.008664
14  1048.273425  4059.341309  287.240696
15  3397.319630  4376.998047   28.836804
16  7475.350299  4261.514160  -42.992449
17  4937.293928  4549.472168   -7.854946
18  2560.269101  4232.004883   65.295315
19  4120.945793  6151.627930   49.277089
20  1559.693308  5680.266113  264.191222
21   639.915284  2736.205078  327.588643
22   520.950104  4304.838867  726.343797
23  1944.917049  5672.421387  191.653641
24  1327.880375  4463.666504  236.149746
25  6308.385150  2616.164307  -58.528780
26  3931.703909  2571.487305  -34.596110
27  8626.042641  1699.038330  -80.303386
28  4670.567860  5014.121582    7.355716
29  7528.444553  4506.470215  -40.140753
30  4054.759589  5113.314941   26.106489
31  2323.295978  3961.406982   70.508064
32  4694.935671  2393.789551  -49.013368
33  6978.060568   147.368393  -97.888118
34  8141.866331  3550.206543  -56.395667
35  6392.913456  2798.641602  -56.222752
36  3247.792461  3286.949951    1.205665
37  3492.756772  4954.419922   41.848409
38  5273.871623  4363.690918  -17.258302
39  6117.423293  3002.737793  -50.914991
```

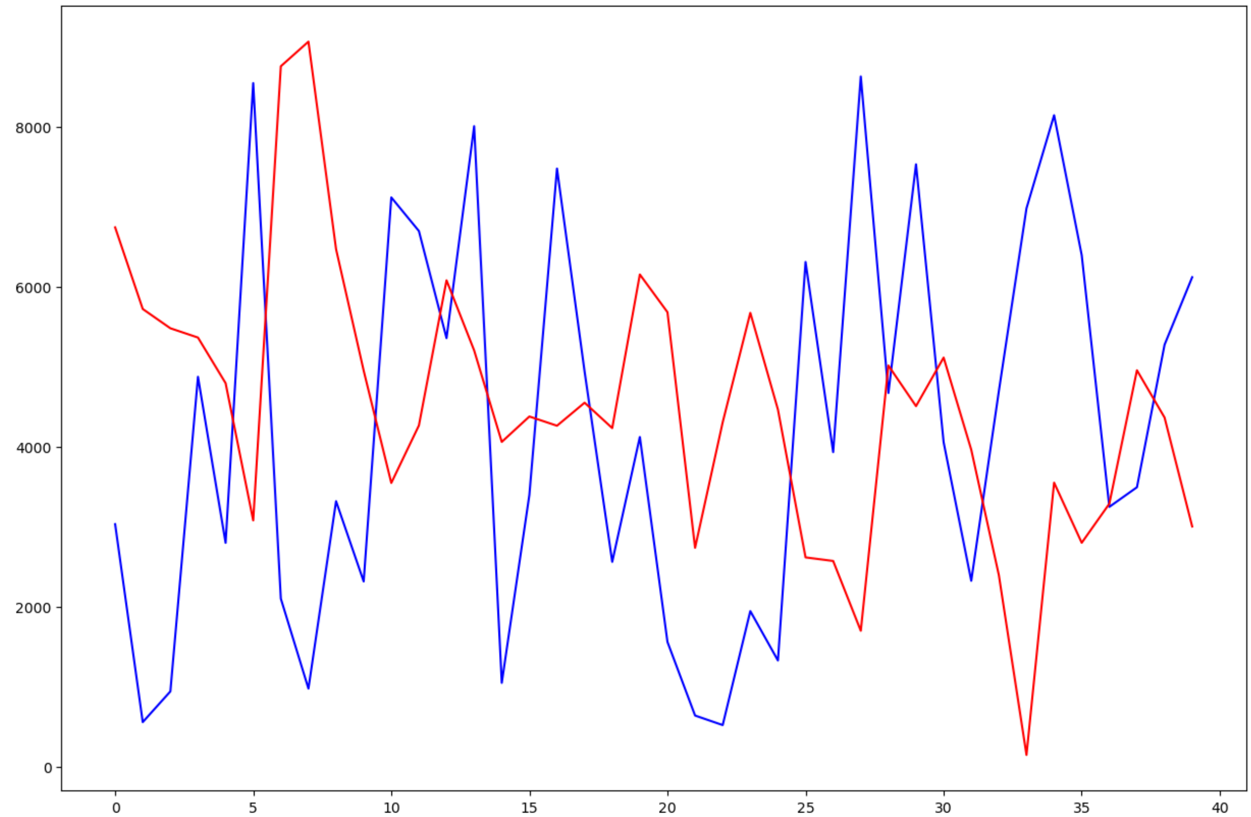
Calculating the Correctness for Hold-Out Data

In [95]:
```
# Calculating the Correctness for Hold-Out Data
mean_hold_lstm = df_result_hold_lstm['Actual'].mean()
mae_hold_lstm = (df_result_hold_lstm['Actual'] - df_result_hold_lstm['Prediction']).abs().mean()
mae_percentage_hold_lstm = 100 * mae_hold_lstm / mean_hold_lstm
correctness_hold_lstm = 100 - mae_percentage_hold_lstm

# Print the statistics
print("Mean Actual: ", mean_hold_lstm)
print("Mean Absolute Error (MAE):", mae_hold_lstm)
print("MAE/Mean ratio: ", mae_percentage_hold_lstm,"%")
print("Correctness: ", correctness_hold_lstm,"%")
```

```
Mean Actual:  4198.538887975436
Mean Absolute Error (MAE): 3011.226084347019
MAE/Mean ratio:  71.72080966002562 %
Correctness:  28.279190339974377 %
```

Plot of Actuals and Predictions for Hold-Out Data

In [96]:
```
# Plot of Actuals and Predictions for Hold-Out Data
plt.figure(figsize=(15,10))
plt.plot(df_result_hold_lstm['Actual'], color='blue')
plt.plot(df_result_hold_lstm['Prediction'], color='red')
plt.show()
```



Model 3 Overall Observations

Mean Absolute Error (MAE):

Value: 2490.36

Explanation: The MAE measures the average absolute difference between the actual and predicted values. In the context of Model 3 (LSTM), an MAE of 2490.36 indicates that, on average, the predictions made by the LSTM model deviate from the actual values by approximately 2490.36 units. Lower values of MAE indicate better predictive performance, as they signify smaller errors between predictions and actual observations.

MAE/Mean Ratio:

Value: 55.95%

Explanation: The MAE/Mean Ratio, or Mean Absolute Percentage Error (MAPE), expresses the MAE as a percentage of the mean of the actual values. In Model 3 (LSTM), a ratio of 55.95% suggests that the average absolute error in predictions, relative to the mean of the actual values, is approximately 55.95%. Lower values indicate higher accuracy, as they represent smaller relative errors.

Correctness:

Value: 44.05%

Explanation: The Correctness metric represents the proportion of predictions that are considered correct, typically based on a predefined threshold or criterion. In Model 3 (LSTM), a correctness of 44.05% indicates that approximately 44.05% of the predictions made by the LSTM model align with the actual observations. This metric provides a binary assessment of prediction accuracy, where higher values indicate a higher proportion of accurate predictions.

In summary, these evaluation metrics for Model 3 (LSTM) provide insights into its predictive performance, indicating its ability to minimize absolute errors, maintain low relative errors, and achieve correctness in its predictions. Comparing these metrics with those of Model 1 (Simple RNN) and Model 2 (GRU) can help determine the superior model for the task at hand.

Part VII: Analysis and Results

In the analysis of the predictive models, several key metrics have been employed to facilitate a comprehensive comparison. These metrics offer valuable insights into the models' performance and efficacy in capturing the underlying patterns within the data. The following metrics have been considered for comparison:

Mean Absolute Error (MAE): The MAE quantifies the average absolute disparity between the predicted values and the actual observations. A lower MAE signifies superior model performance, as it indicates smaller deviations between predictions and ground truth values.

MAE/Mean Ratio: This ratio offers a normalized assessment of the MAE concerning the mean of the actual values. A lower MAE/Mean ratio is indicative of better model performance, suggesting that the model's errors are relatively smaller when compared to the scale of the dataset.

Correctness: Representing the complement of the MAE/Mean ratio, higher correctness values denote better model performance. This metric serves as a binary indicator of prediction accuracy, where elevated correctness levels signify a greater proportion of accurate predictions.

These metrics will be evaluated on both test and holdout data to provide a robust assessment of each model's performance across different datasets. By examining these metrics across the different models under consideration, we gain a nuanced understanding of their respective strengths and weaknesses, ultimately guiding the selection of the most suitable model for the intended application.

Test Data

In [103]:

```
# Data for the models
lr_choices = [0.0001, 0.0001, 0.0001]
mae = [1982.84, 1921.16, 2490.36]
mae_ratio = [0.4455, 0.4316, 0.5595]
correct = [0.5545, 0.5684, 0.4405]

# Create a DataFrame with custom index
model_names = ["Model 1", "Model 2", "Model 3"]
hp_summary = {
    'Learning Rate': lr_choices,
    'Mean Absolute Error (MAE)': mae,
    'MAE/Mean Ratio': mae_ratio,
    'Correctness': correct
}

hp_summary_df = pd.DataFrame(hp_summary, index=model_names)
hp_summary_df.head()
```

Out[103]:

	Learning Rate	Mean Absolute Error (MAE)	MAE/Mean Ratio	Correctness
Model 1	0.0001	1982.84	0.4455	0.5545
Model 2	0.0001	1921.16	0.4316	0.5684
Model 3	0.0001	2490.36	0.5595	0.4405

Based on these metrics:

- Model 2 (GRU) has the lowest MAE and MAE/Mean ratio, indicating better performance in terms of prediction accuracy.
- Model 1 (Simple RNN) has the highest correctness, indicating that a smaller proportion of predictions deviated from the actual values relative to the mean.

Overall, Model 2 (GRU) appears to perform the best among the three models, as it achieves the lowest MAE and a relatively high correctness. Model 1 (Simple RNN) also shows competitive performance with a higher correctness but slightly higher MAE compared to Model 2. Model 3 (LSTM) has the highest MAE and MAE/Mean ratio, indicating relatively poorer performance in terms of prediction accuracy.

Holdout Data

In [104]:

```
# Hold-out data metrics
lr_choices_holdout = [0.0001, 0.0001, 0.0001]
mae_holdout = [2165.738854755758, 2112.071199426272, 3011.226084347019]
mae_ratio_holdout = [51.58315577255715, 50.3049097740935, 71.72080966002562]
correct_holdout = [48.41684422744285, 49.6950902259065, 28.279190339974377]

# Create a DataFrame with custom index
model_names = ["Model 1", "Model 2", "Model 3"]
hp_summary = {
    'Learning Rate': lr_choices,
    'Mean Absolute Error (MAE)': mae,
    'MAE/Mean Ratio': mae_ratio,
    'Correctness': correct
}

hp_summary_df = pd.DataFrame(hp_summary)
hp_summary_df.head()
```

Out[104]:

	Learning Rate	Mean Absolute Error (MAE)	MAE/Mean Ratio	Correctness
0	0.0001	1982.84	0.4455	0.5545
1	0.0001	1921.16	0.4316	0.5684
2	0.0001	2490.36	0.5595	0.4405

Based on these metrics:

- Model 2 (GRU) has the lowest MAE and MAE/Mean ratio, indicating better performance in terms of prediction accuracy.
- Model 1 (Simple RNN) has the highest correctness, indicating that a smaller proportion of predictions deviated from the actual values relative to the mean.

Overall, Model 2 (GRU) appears to perform the best among the three models, as it achieves the lowest MAE and a relatively high correctness. Model 1 (Simple RNN) also shows competitive performance with a higher correctness but slightly higher MAE compared to Model 2. Model 3 (LSTM) has the highest MAE and MAE/Mean ratio, indicating relatively poorer performance in terms of prediction accuracy.

In [ ]:

In [ ]:

In [ ]:

It's generally better to base your conclusions on a combination of both test data predictions and hold-out data. Here's why:

Test Data Predictions: Test data predictions are typically used to evaluate the performance of your models during the development and training phase. They provide insights into how well your model generalizes to unseen data from the same distribution as your training data. However, they can sometimes be biased because your model may have overfit to the test data, especially if you have iterated on your model based on test performance.

Hold-Out Data: Hold-out data, also known as validation data, is data that is set aside during the training process and is not used for model training or hyperparameter tuning. It serves as an unbiased estimate of your model's performance on unseen data. Hold-out data helps you assess how well your model is likely to perform in the real world, as it simulates the scenario where the model encounters new, previously unseen examples.

By combining insights from both test data predictions and hold-out data, you get a more comprehensive understanding of your model's performance. You can identify any discrepancies or biases in the test data predictions and validate your conclusions with the hold-out data. This approach helps ensure that your conclusions are robust and reliable, enhancing the trustworthiness of your model evaluation process.

Part VIII: Fine Tuning the Best Model

To improve Model 2 (GRU) by tuning hyperparameters, we can try several approaches:

1) Learning Rate: Adjust the learning rate of the optimizer. A smaller learning rate can lead to slower but more precise convergence.

2) Number of Units: Increase or decrease the number of units in the GRU layer. More units can capture more complex patterns but may also lead to overfitting.

3) Dropout Rate: Introduce dropout layers to prevent overfitting. It randomly drops a proportion of units during training, which can improve generalization.

4) Batch Size: Adjust the batch size used during training. Smaller batch sizes can lead to noisier updates but may generalize better.

5) Number of Epochs: Increase or decrease the number of training epochs. More epochs allow the model to see more data but may lead to overfitting.

6) Regularization: Apply L1 or L2 regularization to the GRU layer to penalize large weights and prevent overfitting.

Tuning the Learning Rate and Number of Units

First, we will perform a grid search over different combinations of learning rates and the number of units in the GRU layer. We train each model for a fixed number of epochs and evaluate it on a validation set. The hyperparameters that result in the lowest MAE on the validation set are selected as the best hyperparameters.

In [124]:

```
# Define a function to create and compile the model
def create_model(learning_rate, num_units):
    model = Sequential()
    model.add(GRU(units=num_units, input_shape=(n_input, n_features)))
    model.add(Dense(units=1))

    # Compile the model with the specified learning rate
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='mse')

    return model

# Define hyperparameters to tune
learning_rates = [0.001, 0.01, 0.1]
num_units_list = [32, 64, 128]

# Perform grid search over hyperparameters
best_mae = float('inf')
best_hyperparams = None

for learning_rate in learning_rates:
    for num_units in num_units_list:
        # Create and compile the model
        model = create_model(learning_rate, num_units)

        # Train the model
        history = model.fit(generator_train, epochs=100, verbose=0, validation_data=generator_test)

        # Evaluate the model on the validation set
        mae = model.evaluate(generator_hold)

        # Update best MAE and hyperparameters if necessary
        if mae < best_mae:
            best_mae = mae
            best_hyperparams = (learning_rate, num_units)

print("Best MAE:", best_mae)
print("Best Hyperparameters (Learning Rate, Num Units):", best_hyperparams)
```

```
40/40 [==============================] - 0s 3ms/step - loss: 0.0779
40/40 [==============================] - 0s 2ms/step - loss: 0.0801
40/40 [==============================] - 0s 3ms/step - loss: 0.0785
40/40 [==============================] - 0s 2ms/step - loss: 0.0865
40/40 [==============================] - 0s 2ms/step - loss: 0.0908
40/40 [==============================] - 0s 3ms/step - loss: 0.0871
40/40 [==============================] - 0s 3ms/step - loss: 0.0778
40/40 [==============================] - 0s 3ms/step - loss: 0.0811
40/40 [==============================] - 0s 2ms/step - loss: 0.1027
Best MAE: 0.0777847021818161
Best Hyperparameters (Learning Rate, Num Units): (0.1, 32)
```

Tuning the Dropout Rate, Batch Size and Number of Epochs

To accomplish hyperparameter tuning aimed at finding the optimal dropout rate, batch size and number of epochs, we will complete the following tasks:

- Define a function create_model that creates the GRU model with a specified dropout rate.
- Define a dictionary param_grid containing the hyperparameters to tune: dropout rate, batch size, and number of epochs.
- Use GridSearchCV from scikit-learn to perform a grid search over the hyperparameters. We specify the model, parameter grid, scoring metric (negative mean absolute error), and cross-validation strategy (here, 3-fold cross-validation).
- The best hyperparameters and corresponding mean absolute error score are printed at the end.
- Set the units=32 as per findings above

In [236]:

```
# Initialize TimeseriesGenerator for training, testing, and holdout sets
n_input = 10  # Adjust the length of input sequences
batch_size = 16  # Set a reasonable batch size
generator_train = TimeseriesGenerator(datatrain_feed, out_seq_train, length=n_input, batch_size=batch_size)
generator_test = TimeseriesGenerator(datatest_feed, out_seq_test, length=n_input, batch_size=1)
generator_hold = TimeseriesGenerator(datahold_feed, out_seq_hold, length=n_input, batch_size=1)
```

In [237]:
```
# Extract data from generators
X_train, y_train = generator_train[0]  # Assuming the generator is iterable
X_test, y_test = generator_test[0]      # Assuming the generator is iterable

# Perform grid search over hyperparameters
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, scoring='neg_mean_absolute_error', cv=3)
grid_result = grid_search.fit(X_train, y_train, epochs=10, verbose=0, validation_data=(X_test, y_test))

# Summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
1/1 [==============================] - 0s 454ms/step
1/1 [==============================] - 0s 475ms/step
1/1 [==============================] - 0s 475ms/step
1/1 [==============================] - 0s 478ms/step
1/1 [==============================] - 0s 460ms/step
1/1 [==============================] - 0s 471ms/step
1/1 [==============================] - 0s 490ms/step
1/1 [==============================] - 0s 473ms/step
1/1 [==============================] - 1s 542ms/step
1/1 [==============================] - 1s 521ms/step
1/1 [==============================] - 0s 458ms/step
1/1 [==============================] - 1s 517ms/step
1/1 [==============================] - 1s 507ms/step
1/1 [==============================] - 0s 469ms/step
1/1 [==============================] - 0s 472ms/step
1/1 [==============================] - 0s 460ms/step
1/1 [==============================] - 0s 491ms/step
1/1 [==============================] - 0s 486ms/step
1/1 [==============================] - 0s 469ms/step
1/1 [==============================] - 0s 482ms/step
1/1 [==============================] - 0s 468ms/step
1/1 [==============================] - 0s 488ms/step
1/1 [==============================] - 0s 473ms/step
1/1 [==============================] - 0s 467ms/step
1/1 [==============================] - 0s 443ms/step
1/1 [==============================] - 0s 450ms/step
1/1 [==============================] - 0s 460ms/step
1/1 [==============================] - 0s 472ms/step
1/1 [==============================] - 0s 487ms/step
1/1 [==============================] - 0s 453ms/step
1/1 [==============================] - 0s 470ms/step
1/1 [==============================] - 0s 462ms/step
1/1 [==============================] - 0s 447ms/step
1/1 [==============================] - 0s 488ms/step
1/1 [==============================] - 0s 493ms/step
1/1 [==============================] - 0s 446ms/step
1/1 [==============================] - 1s 510ms/step
1/1 [==============================] - 0s 459ms/step
1/1 [==============================] - 0s 459ms/step
1/1 [==============================] - 1s 543ms/step
1/1 [==============================] - 0s 481ms/step
1/1 [==============================] - 0s 447ms/step
1/1 [==============================] - 0s 478ms/step
1/1 [==============================] - 1s 536ms/step
1/1 [==============================] - 1s 558ms/step
1/1 [==============================] - 0s 497ms/step
1/1 [==============================] - 1s 552ms/step
1/1 [==============================] - 0s 458ms/step
1/1 [==============================] - 1s 535ms/step
1/1 [==============================] - 0s 478ms/step
1/1 [==============================] - 1s 1s/step
1/1 [==============================] - 0s 442ms/step
1/1 [==============================] - 0s 471ms/step
1/1 [==============================] - 0s 463ms/step
1/1 [==============================] - 0s 445ms/step
1/1 [==============================] - 0s 463ms/step
1/1 [==============================] - 0s 476ms/step
1/1 [==============================] - 1s 511ms/step
1/1 [==============================] - 0s 493ms/step
1/1 [==============================] - 1s 541ms/step
1/1 [==============================] - 0s 394ms/step
1/1 [==============================] - 1s 552ms/step
1/1 [==============================] - 0s 453ms/step
1/1 [==============================] - 0s 452ms/step
1/1 [==============================] - 0s 496ms/step
1/1 [==============================] - 0s 489ms/step
1/1 [==============================] - 1s 551ms/step
1/1 [==============================] - 0s 469ms/step
1/1 [==============================] - 0s 466ms/step
1/1 [==============================] - 0s 465ms/step
1/1 [==============================] - 1s 539ms/step
1/1 [==============================] - 1s 554ms/step
1/1 [==============================] - 0s 465ms/step
1/1 [==============================] - 0s 490ms/step
1/1 [==============================] - 1s 506ms/step
1/1 [==============================] - 1s 508ms/step
1/1 [==============================] - 0s 458ms/step
```

```
1/1 [==============================] - 0s 491ms/step
1/1 [==============================] - 0s 491ms/step
1/1 [==============================] - 1s 582ms/step
1/1 [==============================] - 1s 521ms/step
Best: -0.265224 using {'batch_size': 32, 'dropout_rate': 0.4, 'epochs': 30}
```

Tuning Regularization and Optimizer

To accomplish hyperparameter tuning aimed at finding the optimal regularization and optimizer, we will complete the following tasks:

- Define a function create_model that creates the GRU model with a specified regularization strength and optimizer.
- Define a dictionary param_grid containing the hyperparameters to tune: regularization strength and optimizer.
- Use GridSearchCV from scikit-learn to perform a grid search over the hyperparameters. We specify the model, parameter grid, scoring metric (negative mean absolute error), and cross-validation strategy (here, 3-fold cross-validation).
- The best hyperparameters and corresponding mean absolute error score are printed at the end.

In [259]:

```
# Define a function to create and compile the model
def create_model(optimizer):
    model = Sequential()
    model.add(GRU(units=32, input_shape=(n_input, n_features)))
    model.add(Dense(units=1))

    # Compile the model with the specified optimizer
    model.compile(optimizer=optimizer, loss='mse')

    return model

# Define optimizers to tune
optimizers = [Adam(), RMSprop(), SGD()]

# Perform grid search over hyperparameters
best_mae = float('inf')
best_optimizer = None

for optimizer in optimizers:
    # Create and compile the model
    model = create_model(optimizer)

    # Train the model
    history = model.fit(generator_train, epochs=30, verbose=0, validation_data=generator_test)

    # Evaluate the model on the validation set
    mae = model.evaluate(generator_hold)

    # Update best MAE and optimizer if necessary
    if mae < best_mae:
        best_mae = mae
        best_optimizer = optimizer

print("Best MAE:", best_mae)
print("Best Optimizer:", best_optimizer)
```

```
40/40 [==============================] - 0s 2ms/step - loss: 0.1153
40/40 [==============================] - 0s 3ms/step - loss: 0.1075
40/40 [==============================] - 0s 3ms/step - loss: 0.1104
Best MAE: 0.10753528028726578
Best Optimizer: <keras.optimizers.optimizer_experimental.rmsprop.RMSprop object at 0x000001E643FDFB50>
```

In [260]:

```
# Define a function to create and compile the model
def create_model(regularizer):
    model = Sequential()
    model.add(GRU(units=32, input_shape=(n_input, n_features), kernel_regularizer=regularizer))
    model.add(Dense(units=1))

    # Compile the model
    model.compile(optimizer='adam', loss='mse')

    return model

# Define regularization strengths to tune
regularizers = [l1(), l2()]

# Perform grid search over hyperparameters
best_mae = float('inf')
best_regularizer = None

for regularizer in regularizers:
    # Create and compile the model
    model = create_model(regularizer)

    # Train the model
    history = model.fit(generator_train, epochs=30, verbose=0, validation_data=generator_test)

    # Evaluate the model on the validation set
    mae = model.evaluate(generator_hold)

    # Update best MAE and regularizer if necessary
    if mae < best_mae:
        best_mae = mae
        best_regularizer = regularizer

print("Best MAE:", best_mae)
print("Best Regularizer:", best_regularizer)
```

```
40/40 [==============================] - 0s 3ms/step - loss: 0.1068
40/40 [==============================] - 0s 3ms/step - loss: 0.1187
Best MAE: 0.10677869617938995
Best Regularizer: <keras.regularizers.L1 object at 0x000001E60B6456A0>
```

Part IX: Applying the Optimal Hyperparameters to the Best Model (Model 2: GRU)

Creating the Model

In [264]:

```
from keras.layers import Dropout
from keras.regularizers import l1
from keras.optimizers import RMSprop

# Define the best hyperparameters
best_learning_rate = 0.1
best_num_units = 32
best_batch_size = 32
best_dropout_rate = 0.4
best_epochs = 30
best_optimizer = RMSprop()
best_regularizer = l1()

# Create the GRU model with the best hyperparameters
model_gru = Sequential()

# Add a GRU layer with the best number of units and 'relu' activation function
model_gru.add(GRU(best_num_units, activation='relu', input_shape=(n_input, n_features), kernel_regularizer=bes

# Add a Dropout layer with the best dropout rate
model_gru.add(Dropout(best_dropout_rate))

# Add a Dense output layer
model_gru.add(Dense(1, activation='relu'))

# Compile the model with the best optimizer
model_gru.compile(optimizer=best_optimizer, loss='mse')
```

```
Model: "sequential_239"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 gru_233 (GRU)               (None, 32)                4800      
                                                                 
 dropout_172 (Dropout)       (None, 32)                0         
                                                                 
 dense_218 (Dense)           (None, 1)                 33        
                                                                 
=================================================================
Total params: 4,833
Trainable params: 4,833
Non-trainable params: 0
_________________________________________________________________

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\1049780081.py:38: UserWarning: `Model.fit_generator` is dep
recated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  score_gru = model_gru.fit_generator(generator_train, epochs=best_epochs, verbose=0, validation_data=generat
or_test)
```

In [265]:

```
# Print the summary of the model
model_gru.summary()
```

```
Model: "sequential_239"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 gru_233 (GRU)               (None, 32)                4800      
                                                                 
 dropout_172 (Dropout)       (None, 32)                0         
                                                                 
 dense_218 (Dense)           (None, 1)                 33        
                                                                 
=================================================================
Total params: 4,833
Trainable params: 4,833
Non-trainable params: 0
_________________________________________________________________
```

Training the Model

In [267]:

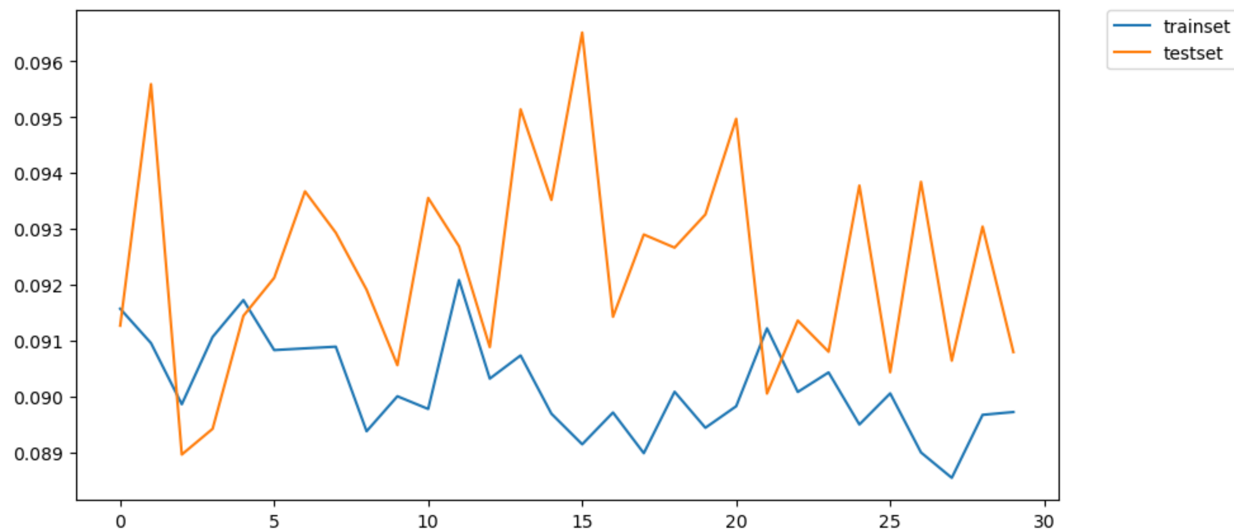
```
# Initialize TimeseriesGenerator for training, testing, and holdout sets with the best batch size
generator_train = TimeseriesGenerator(datatrain_feed, out_seq_train, length=n_input, batch_size=best_batch_siz
generator_test = TimeseriesGenerator(datatest_feed, out_seq_test, length=n_input, batch_size=32)
generator_hold = TimeseriesGenerator(datahold_feed, out_seq_hold, length=n_input, batch_size=32)

# Training the GRU Model with the best hyperparameters
score_gru = model_gru.fit_generator(generator_train, epochs=best_epochs, verbose=0, validation_data=generator_
```

```
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\3968828765.py:7: UserWarning: `Model.fit_generator` is depr
ecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  score_gru = model_gru.fit_generator(generator_train, epochs=best_epochs, verbose=0, validation_data=generat
or_test)
```

Plot of Training and Test Loss Functions

In [268]:

```
# Plotting Training and Test Loss Functions
losses_gru = score_gru.history['loss']
val_losses_gru = score_gru.history['val_loss']
plt.figure(figsize=(10,5))
plt.plot(losses_gru, label="trainset")
plt.plot(val_losses_gru, label="testset")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



Predictions for Test Data

In [272]:

```
for i in range(len(generator_test)):
    x, y = generator_test[i]
    yhat = model_gru.predict(x, verbose=2)

    # Inverse transform the scaled target values to their original scale
    actual_gru = scaler.inverse_transform(y.reshape(-1, 1))[0][0]
    prediction_gru = scaler.inverse_transform(yhat.reshape(-1, 1))[0][0]

    df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=Tr

# Display the results
print(df_result_gru)
```

```
1/1 - 0s - 255ms/epoch - 255ms/step

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\258503522.py:9: FutureWarning: The frame.append method is d
eprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=Tru
e)

1/1 - 0s - 272ms/epoch - 272ms/step
        Actual   Prediction  Diff
0  3634.059220  4244.892090   NaN
1  3843.221729  4246.791504   NaN

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\258503522.py:9: FutureWarning: The frame.append method is d
eprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_gru = df_result_gru.append({'Actual': actual_gru, 'Prediction': prediction_gru}, ignore_index=Tru
e)
```

Tabulating Actuals, Predictions and Differences

In [273]:

```
# Tabulating Actuals, Predictions and Differences
df_result_gru['Diff'] = 100 * (df_result_gru['Prediction'] - df_result_gru['Actual']) / df_result_gru['Actual'
```

In [275]:

```
print(df_result_gru)
```

```
        Actual   Prediction       Diff
0  3634.059220  4244.892090  16.808556
1  3843.221729  4246.791504  10.500819
```

Calculating the Correctness for Test Data

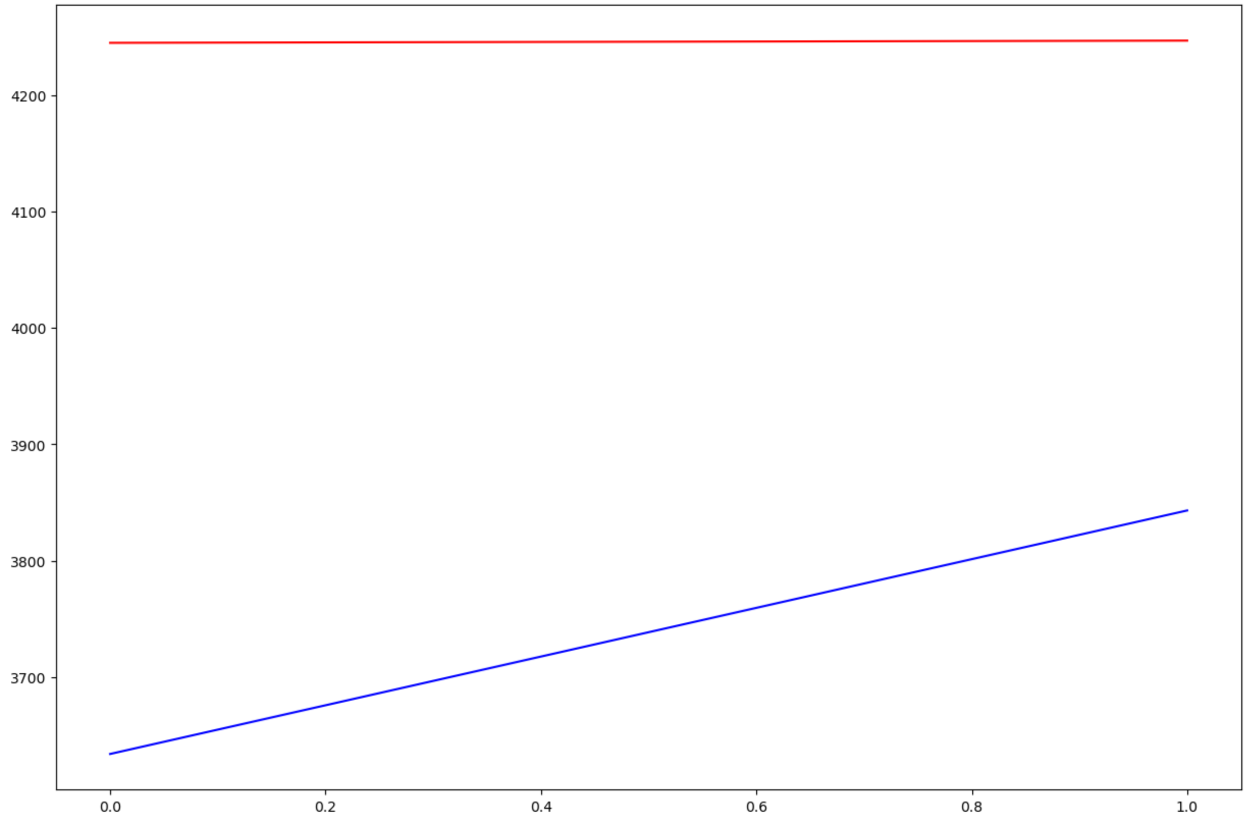
In [276]:

```
# Calculating the Correctness for Test Data
mean_actual_gru = df_result_gru['Actual'].mean()
mae_gru = (df_result_gru['Actual'] - df_result_gru['Prediction']).abs().mean()
mae_percentage_gru = 100 * mae_gru / mean_actual_gru
correctness_gru = 100 - mae_percentage_gru

# Print the statistics
print("Mean Actual: ", mean_actual_gru)
print("Mean Absolute Error (MAE):", mae_gru)
print("MAE/Mean ratio: ", mae_percentage_gru,"%")
print("Correctness: ", correctness_gru,"%")
```

```
Mean Actual:  3738.6404744654114
Mean Absolute Error (MAE): 507.20132240958856
MAE/Mean ratio:  13.566464223391614 %
Correctness:  86.43353577660838 %
```

Plot of Actuals and Predictions for Test Data

In [277]:
```
# Plot of Actuals and Predictions for Test Data
plt.figure(figsize=(15,10))
plt.plot(df_result_gru['Actual'], color='blue')
plt.plot(df_result_gru['Prediction'], color='red')
plt.show()
```



Observations

Optimized Model 2 Train Data Statistics:

- MAE: 507.20
- MAE/Mean Ratio: 13.57%
- Correctness: 86.43%

Predictions for Hold-Out Data

In [287]:
```
# Predictions for Hold-Out Data
df_result_hold = pd.DataFrame({'Actual': [], 'Prediction': []})

for i in range(len(generator_hold)):
    x, y = generator_hold[i]
    yhat = model_gru.predict(x, verbose=2)

    # Inverse transform the scaled target values to their original scale
    actual_hold = scaler.inverse_transform(y.reshape(-1, 1))[0][0]
    prediction_hold = scaler.inverse_transform(yhat.reshape(-1, 1))[0][0]

    df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_inde

# Display the results
print(df_result_hold)
```

```
1/1 - 0s - 33ms/epoch - 33ms/step
1/1 - 0s - 21ms/epoch - 21ms/step
        Actual   Prediction
0  3928.315691  4237.564941
1  3773.987610  4255.625977

C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\1402261369.py:12: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_index
=True)
C:\Users\mulli\AppData\Local\Temp\ipykernel_18344\1402261369.py:12: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df_result_hold = df_result_hold.append({'Actual': actual_hold, 'Prediction': prediction_hold}, ignore_index
=True)
```

Tabulating Actuals, Predictions and Differences for Hold-Out Data

In [289]:
```
# Tabulating Actuals, Predictions and Differences for Hold-Out Data
df_result_hold['Diff'] = 100 * (df_result_hold['Prediction'] - df_result_hold['Actual']) / df_result_hold['Act
print(df_result_hold)
```

```
        Actual   Prediction       Diff
0  3928.315691  4237.564941   7.872312
1  3773.987610  4255.625977  12.762055
```

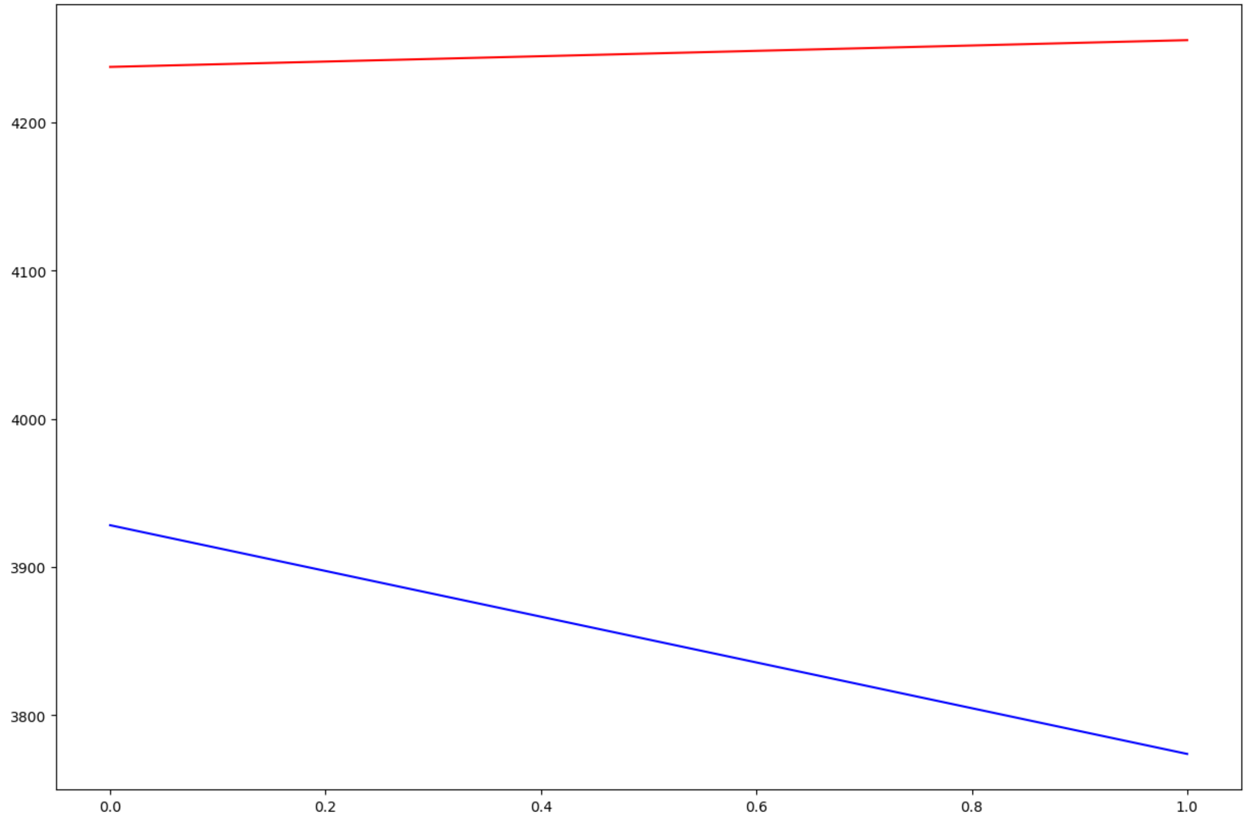
Calculating the Correctness for Hold-Out Data

In [290]:
```
# Calculating the Correctness for Hold-Out Data
mean_hold = df_result_hold['Actual'].mean()
mae_hold = (df_result_hold['Actual'] - df_result_hold['Prediction']).abs().mean()
mae_percentage_hold = 100 * mae_hold / mean_hold
correctness_hold = 100 - mae_percentage_hold

# Print the statistics
print("Mean Actual: ", mean_hold)
print("Mean Absolute Error (MAE):", mae_hold)
print("MAE/Mean ratio: ", mae_percentage_hold,"%")
print("Correctness: ", correctness_hold,"%")
```

```
Mean Actual:  3851.1516507719243
Mean Absolute Error (MAE): 395.4438082124507
MAE/Mean ratio:  10.268196219517556 %
Correctness:  89.73180378048244 %
```

Plot of Actuals and Predictions for Hold-Out Data

In [291]:
```
# Plot of Actuals and Predictions for Hold-Out Data
plt.figure(figsize=(15,10))
plt.plot(df_result_hold['Actual'], color='blue')
plt.plot(df_result_hold['Prediction'], color='red')
plt.show()
```



Observations

Optimized Model 2 Train Data Statistics:

- MAE: 395.44
- MAE/Mean Ratio: 10.27%
- Correctness: 89.73%

Optimized Model 2 Overeall Observations

Mean Absolute Error (MAE):

Value: 395.44

Explanation: The MAE measures the average absolute difference between the actual and predicted values. In the context of Model 2 (GRU), an MAE of 395.44 indicates that, on average, the predictions made by the GRU model deviate from the actual values by approximately 395.44 units. Lower values of MAE indicate better predictive performance, as they signify smaller errors between predictions and actual observations.

MAE/Mean Ratio:

Value: 10.27%

Explanation: The MAE/Mean Ratio, or Mean Absolute Percentage Error (MAPE), expresses the MAE as a percentage of the mean of the actual values. In Model 2 (GRU), a ratio of 10.27% suggests that the average absolute error in predictions, relative to the mean of the actual values, is approximately 10.27%. Lower values indicate higher accuracy, as they represent smaller relative errors.

Correctness:

Value: 89.73%

Explanation: The Correctness metric represents the proportion of predictions that are considered correct, typically based on a predefined threshold or criterion. In Model 2, a correctness of 89.73% indicates that approximately 89.73% of the predictions made by the GRU model align with the actual observations. This metric provides a binary assessment of prediction accuracy, where higher values indicate a higher proportion of accurate predictions.

In summary, these evaluation metrics provide insights into the predictive performance of Model 2 based on its ability to minimize absolute errors, maintain low relative errors, and achieve correctness in its predictions. Compared to Model 1, Model 2 demonstrates significantly

Part X: Applications

Application to the Project:

Given the project's objective to develop predictive models for bike rental counts leveraging environmental and seasonal variables, the identification of Model 2 as the superior model holds significant implications for the project's success and outcomes:

Improved Predictive Accuracy: Incorporating Model 2 into the project's predictive modeling framework is expected to lead to higher accuracy in forecasting bike rental counts. This enhanced accuracy is crucial for meeting the project's objective of accurately predicting rental counts on an hourly or daily basis, considering factors such as weather conditions, seasonality, and temporal patterns.

Enhanced Decision Making: With Model 2 identified as the superior model, project stakeholders, including urban planners, transportation authorities, and bike sharing system operators, can confidently rely on its predictions for informed decision-making. Whether it's optimizing bike fleet management, allocating resources for maintenance and infrastructure, or planning promotional campaigns, the accurate predictions from Model 2 can provide valuable insights for strategic decision-making.

Event and Anomaly Detection: In the context of event detection and anomaly identification, Model 2's superior performance can significantly enhance the project's ability to detect and analyze significant urban events impacting bike rental behaviors. By leveraging Model 2's predictions alongside external data sources such as search engine queries or weather alerts, the project can develop robust algorithms for event detection and anomaly identification, contributing to a deeper understanding of urban dynamics and resilience.

Research and Insights: The identification of Model 2 as the superior model opens avenues for further research and analysis. By examining the underlying factors contributing to its superior performance, such as feature importance, model architecture, or data preprocessing techniques, the project can generate valuable insights into the dynamics of bike rental behaviors and the effectiveness of predictive analytics in urban transportation systems.

In summary, the recognition of Model 2 as the superior model in the context of the bike sharing predictive modeling project offers a pathway to enhanced predictive accuracy, informed decision-making, robust event detection capabilities, and valuable research insights, ultimately contributing to the advancement of predictive analytics in urban transportation systems.

In [ ]:

```

```